II Arbitrarily Large Data

Every data definition in Fixed-Size Data describes data of a fixed size. To us, Boolean values, numbers, strings, and images are atomic; computer scientists say they have a size of one unit. With a structure, you compose a fixed number of pieces of data. Even if you use the language of data definitions to create deeply nested structures, you always know the exact number of atomic pieces of data in any specific instance. Many programming problems, however, deal with an undetermined number of pieces of information that must be processed as one piece of data. For example, one program may have to compute the average of a bunch of numbers and another may have to keep track of an arbitrary number of objects in an interactive game. Regardless, it is impossible with your knowledge to formulate a data definition that can represent this kind of information as data.

This part revises the language of data definitions so that it becomes possible to describe data of (finite but) arbitrary size. For a concrete illustration, the first half of this part deals with lists, a form of data that appears in most modern programming languages. In parallel with the extended language of data definitions, this part also revises the design recipe to cope with such data definitions. The latter chapters demonstrate how these data definitions and the revised design recipe work in a variety of contexts.

8 Lists

You have probably not encountered self-referential definitions before. Your English teachers certainly stay away from these, and many mathematics courses are vague when it comes to such definitions. Programmers cannot afford to be vague. Their work requires precision. While a definition may in general contain several references to itself, this chapter presents useful examples that need just one, starting with the one for lists.

The introduction of lists also spices up the kind of applications we can study. While this chapter carefully builds up your intuition with examples, it also motivates the revision of the design recipe in the next chapter, which explains how to systematically create functions that deal with self-referential data definitions.

8.1 Creating Lists

All of us make lists all the time. Before we go grocery shopping, we write down a list of items we wish to purchase. Some people write down a to-do list every morning. During December, many children prepare Christmas wish lists. To plan a party, we make a list of invitees. Arranging information in the form of lists is an ubiquitous part of our life.

Given that information comes in the shape of lists, we must clearly learn how to represent such lists as BSL data. Indeed, because lists are so important, BSL comes with built-in support for creating and manipulating lists, similar to the support for Cartesian points (`posn`). In contrast to points, the data definition for lists is always left to you. But first things first. We start with the creation of lists.

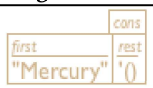
When we form a list, we always start out with the empty list. In BSL, we represent the *empty* list with

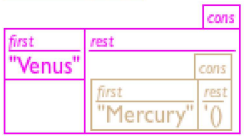
```
'()
```

which is pronounced “empty,” short for “empty list.” Like `#true` or `5`, `'()` is just a constant. When we add something to a list, we construct another list; in BSL, the `cons` operation serves this purpose. For example,

```
(cons "Mercury" '())
```

constructs a list from the `'()` list and the string `"Mercury"`. Figure 44 presents this list in the same pictorial manner we used for structures. The box for `cons` has two fields: `first` and `rest`. In this specific example the `first` field contains `"Mercury"` and the `rest` field contains `'()`.

list	diagram
`(cons "Mercury"` `  '())`	

`(cons "Venus"`
`  (cons "Mercury"`
`  '()))`
	

`(cons "Earth"`
`  (cons "Venus"`
`  (cons "Mercury"`	

```
'())))
```

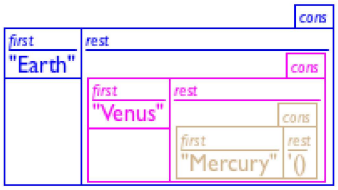


Figure 44: Building a list

Once we have a list with one item in it, we can construct lists with two items by using `cons` again. Here is one:

```
(cons "Venus" (cons "Mercury" '()))
```

And here is another:

```
(cons "Earth" (cons "Mercury" '()))
```

The middle row of figure 44 shows how you can imagine lists that contain two items. It is also a box of two fields, but this time the `rest` field contains a box. Indeed, it contains the box from the top row of the same figure.

Finally, we construct a list with three items:

```
(cons "Earth" (cons "Venus" (cons "Mercury" '())))
```

The last row of figure 44 illustrates the list with three items. Its `rest` field contains a box that contains a box again. So, as we create lists we put boxes into boxes into boxes, and so on. While this may appear strange at first glance, it is just like a set of Chinese gift boxes or a set of nested drinking cups, which we sometimes get for birthdays. The only difference is that BSL programs can nest lists much deeper than any artist could nest physical boxes.

```
(cons "Earth"
  (cons "Venus"
    (cons "Mercury"
      '())))
```

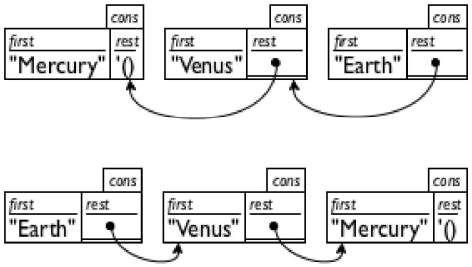


Figure 45: Drawing a list

Because even good artists would have problems with drawing deeply nested structures, computer scientists resort to box-and-arrow diagrams instead. Figure 45 illustrates how to rearrange the last row of figure 44. Each `cons` structure becomes a separate box. If the rest field is too complex to be drawn inside of the box, we draw a bullet instead and a line with an arrow to the box that it contains. Depending on how the boxes are arranged, you get two kinds of diagrams. The first, displayed in the top row of figure 45, lists the boxes in the order in which they are created. The second, displayed in the bottom row, lists the boxes in the order in which they are `cons`ed together. Hence the second diagram immediately tells you what `first` would have produced when applied to the list, no matter how long the list is. For this reason, programmers prefer the second arrangement.

Exercise 129. Create BSL lists that represent

1. a list of celestial bodies, say, at least all the planets in our solar system;

2. a list of items for a meal, for example, steak, french fries, beans, bread, water, Brie cheese, and ice cream; and

3. a list of colors.

Sketch some box representations of these lists, similar to those in figures 44 and 45. Which of the sketches do you like better?

You can also make lists of numbers. Here is a list with the 10 digits:

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
```

```
              (cons 7
                (cons 8
                  (cons 9 '())))))))))))
```

To build this list requires 10 list constructions and one `'()`. For a list of three arbitrary numbers, for example,

```
(cons pi
  (cons e
    (cons -22.3 '())))
```

we need three `cons`es.

In general a list does not have to contain values of one kind, but may contain arbitrary values:

```
(cons "Robbie Round"
  (cons 3
    (cons #true
      '())))
```

The first item of this list is a string, the second one is a number, and the last one a Boolean. You may consider this list as the representation of a personnel record with three pieces of data: the name of the employee, the number of years spent at the company, and whether the employee has health insurance through the company. Or, you could think of it as representing a virtual player in some game. Without a data definition, you just can't know what data is all about.

> Then again, if this list is supposed to represent a record with a fixed number of pieces, use a structure type instead.

Here is a first data definition that involves `cons`:

```
; A 3LON is a list of three numbers:
;   (cons Number (cons Number (cons Number '())))
; interpretation a point in 3-dimensional space
```

Of course, this data definition uses `cons` like others use constructors for structure instances, and in a sense, `cons` is just a special constructor. What this data definition fails to demonstrate is how to form lists of arbitrary length: lists that contain nothing, one item, two items, ten items, or perhaps 1,438,901 items.

So let's try again:

```
; A List-of-names is one of:
; – '()
; – (cons String List-of-names)
; interpretation a list of invitees, by last name
```

Take a deep breath, read it again. This data definition is one of the most unusual definitions you have ever encountered—you have never before seen a definition that refers to itself. It isn't even clear whether it makes sense. After all, if you had told your English teacher that "a table is a table" defines the word "table," the most likely response would have been "Nonsense!" because a self-referential definition doesn't explain what a word means.

In computer science and in programming, though, self-referential definitions play a central role, and with some care, such definitions actually do make sense. Here "making sense" means that we can use the data definition for what it is intended for, namely, to generate examples of data that belong to the class that is being defined or to check whether some given piece of data belongs to the defined class of data. From this perspective, the definition of List-of-names makes complete sense. At a minimum, we can generate `'()` as one example, using the first clause in the itemization. Given `'()` as an element of List-of-names, it is easy to make a second one:

```
(cons "Findler" '())
```

Here we are using a String and the only list from List-of-names to generate a piece of data according to the second clause in the itemization. With the same rule, we can generate many more lists of this kind:

```
(cons "Flatt" '())
(cons "Felleisen" '())
(cons "Krishnamurthi" '())
```

And while these lists all contain one name (represented as a String), it is actually possible to use the second line of the data definition to create lists with more names in them:

```
(cons "Felleisen" (cons "Findler" '()))
```

This piece of data belongs to List-of-names because `"Felleisen"` is a String and `(cons "Findler" '())` is a confirmed List-of-names.

Exercise 130. Create an element of List-of-names that contains five Strings. Sketch a box representation of the list similar to those found in figure 44.

Explain why

```
(cons "1" (cons "2" '()))
```

is an element of List-of-names and why `(cons 2 '())` isn't.

Exercise 131. Provide a data definition for representing lists of Boolean values. The class contains all arbitrarily long lists of Booleans.

8.2 What Is `'()`, What Is `cons`

Let's step back for a moment and take a close look at `'()` and `cons`. As mentioned, `'()` is just a constant. When compared to constants such as `5` or `"this is a string"`, it looks more like a function name or a variable; but when compared with `#true` and `#false`, it should be easy to see that it really is just BSL's representation for empty lists.

As for our evaluation rules, `'()` is a new kind of atomic value, distinct from any other kind: numbers, Booleans, strings, and so on. It also isn't a compound value, like Posns. Indeed, `'()` is so unique it belongs in a class of values all by itself. As such, this class of values comes with a predicate that recognizes only `'()` and nothing else:

```
; Any -> Boolean
; is the given value '()
(define (empty? x) ...)
```

Like all predicates, `empty?` can be applied to any value from the universe of BSL values. It produces `#true` precisely when it is applied to `'()`:

```
> (empty? '())
#true
> (empty? 5)
#false
> (empty? "hello world")
#false
> (empty? (cons 1 '()))
#false
> (empty? (make-posn 0 0))
#false
```

Next we turn to `cons`. Everything we have seen so far suggests that `cons` is a constructor just like those introduced by structure type definitions. More precisely, `cons` appears to be the constructor for a two-field structure: the first one for any kind of value and the second one for any list-like value. The following definitions translate this idea into BSL:

```
(define-struct pair [left right])
; A ConsPair is a structure:
;   (make-pair Any Any).

; Any Any -> ConsPair
(define (our-cons a-value a-list)
  (make-pair a-value a-list))
```

The only catch is that `our-cons` accepts all possible BSL values for its second argument and `cons` doesn't, as the following experiment validates:

```
> (cons 1 2)
cons:second argument must be a list, but received 1 and 2
```

Put differently, `cons` is really a checked function, the kind discussed in Itemizations and Structures, which suggests the following refinement:

```
; A ConsOrEmpty is one of:
; – '()
; – (make-pair Any ConsOrEmpty)
; interpretation ConsOrEmpty is the class of all lists

; Any Any -> ConsOrEmpty
(define (our-cons a-value a-list)
  (cond
    [(empty? a-list) (make-pair a-value a-list)]
```

```
    [(pair? a-list) (make-pair a-value a-list)]
    [else (error "cons: second argument ...")]))
```

If `cons` is a checked constructor function, you may be wondering how to extract the pieces from the resulting structure. After all, Adding Structure says that programming with structures requires selectors. Since a `cons` structure has two fields, there are two selectors: `first` and `rest`. They are also easily defined in terms of our `pair` structure:

```
; ConsOrEmpty -> Any
; extracts the left part of the given pair
(define (our-first a-list)
  (if (empty? a-list)
      (error 'our-first "...")
      (pair-left a-list)))
```

Stop! Define `our-rest`.

If your program can access the structure type definition for `pair`, it is easy to create `pairs` that don't contain `'()` or another `pair` in the `right` field. Whether such bad instances are created intentionally or accidentally, they tend to break functions and programs in strange ways. BSL therefore hides the actual structure type definition for `cons` to avoid these problems. Local Definitions demonstrates one way that your programs can hide such definitions, too, but for now, you don't need this power.

`'()`	a special value, mostly to represent the empty list
`empty?`	a predicate to recognize `'()` and nothing else
`cons`	a checked constructor to create two-field instances
`first`	the selector to extract the last item added
`rest`	the selector to extract the extended list
`cons?`	a predicate to recognize instances of `cons`

Figure 46: List primitives

Figure 46 summarizes this section. The key insight is that `'()` is a unique value and that `cons` is a checked constructor that produces list values. Furthermore, `first`, `rest`, and `cons?` are merely distinct names for the usual predicate and selectors. What this chapter teaches, then, is **not** a new way of creating data but **a new way of formulating data definitions**.

8.3 Programming with Lists

Say you're keeping track of your friends with some list, and say the list has grown so large that you need a program to determine whether some name is on the list. To make this idea concrete, let's state it as an exercise:

> Here we use the word "friend" in the sense of social networks, not the real world.

> **Sample Problem** You are working on the contact list for some new cell phone. The phone's owner updates and consults this list on various occasions. For now, you are assigned the task of designing a function that consumes this list of contacts and determines whether it contains the name "Flatt."

Once we have a solution to this sample problem, we will generalize it to a function that finds any name on a list.

The data definition for List-of-names from the preceding is appropriate for representing the list of names that the function is to search. Next we turn to the header material:

```
; List-of-names -> Boolean
; determines whether "Flatt" is on a-list-of-names
(define (contains-flatt? a-list-of-names)
  #false)
```

While `a-list-of-names` is a good name for the list of names that the function consumes, it is a mouthful and we therefore shorten it to `alon`.

Following the general design recipe, we next make up some examples that illustrate the purpose of the function. First, we determine the output for the simplest input: `'()`. Since this list does not contain any strings, it certainly does not contain `"Flatt"`:

```
(check-expect (contains-flatt? '()) #false)
```

Then we consider lists with a single item. Here are two examples:

```
(check-expect (contains-flatt? (cons "Find" '()))
              #false)
(check-expect (contains-flatt? (cons "Flatt" '())))
```

```
              #true)
```

In the first case, the answer is `#false` because the single item on the list is not `"Flatt"`; in the second case, the single item is `"Flatt"`, so the answer is `#true`. Finally, here is a more general example:

```
(check-expect
  (contains-flatt?
    (cons "A" (cons "Flatt" (cons "C" '()))))
  #true)
```

Again, the answer case must be `#true` because the list contains `"Flatt"`. Stop! Make a general example for which the answer must be `#false`.

Take a breath. Run the program. The header is a "dummy" definition for the function; you have some examples; they have been turned into tests; and best of all, some of them actually succeed. They succeed for the wrong reason but succeed they do. If things make sense now, read on.

The fourth step is to design a function template that matches the data definition. Since the data definition for lists of strings has two clauses, the function's body must be a `cond` expression with two clauses. The two conditions determine which of the two kinds of lists the function received:

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) ...]
    [(cons? alon) ...]))
```

Instead of `(cons? alon)`, we could use `else` in the second clause.

We can add one more hint to the template by studying each clause of the `cond` expression in turn. Specifically, recall that the design recipe suggests annotating each clause with selector expressions if the corresponding class of inputs consists of compounds. In our case, we know that `'()` does not have compounds, so there are no components. Otherwise the list is constructed from a string and another list of strings, and we remind ourselves of this fact by adding `(first alon)` and `(rest alon)` to the template:

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) ...]
    [(cons? alon)
     (... (first alon) ... (rest alon) ...)]))
```

Now it is time to switch to the programming task proper, the fifth step of our design recipe. It starts from a template and deals with each `cond` clause separately. If `(empty? alon)` is true, the input is the empty list, in which case the function must produce the result `#false`. In the second case, `(cons? alon)` is true. The annotations in the template remind us that there is a first string and then the rest of the list. So let's consider an example that falls into this category:

```
(cons "A"
  (cons ...
        ... '()))
```

The function, like a person, must compare the first item with `"Flatt"`. In this example, the first one is `"A"` and not `"Flatt"`, so the comparison yields `#false`. If we had considered some other example instead, say,

```
(cons "Flatt"
  (cons ...
        ... '()))
```

the function would determine that the first item on the input is `"Flatt"`, and would therefore respond with `#true`. This implies that the second line in the `cond` expression should contain an expression that compares the first name on the list with `"Flatt"`:

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
     (... (string=? (first alon) "Flatt")
      ... (rest alon) ...)]))
```

Furthermore, if the comparison yields `#true`, the function must produce `#true`. If the comparison yields `#false`, we are left with another list of strings: `(rest alon)`. Clearly, the function can't know the final answer in this case, because the answer depends on what "..." represents. Put differently, if the first item is not `"Flatt"`, we need some way to check whether the rest of the list contains `"Flatt"`.

Fortunately, we have `contains-flatt?` and it fits the bill. According to its purpose statement, it determines whether a list contains `"Flatt"`. The statement implies that `(contains-flatt? l)` tells us whether the list of strings `l` contains `"Flatt"`. And, in the same vein, `(contains-flatt? (rest alon))` determines whether `"Flatt"` is a member of `(rest alon)`, which is precisely what we need to know.

In short, the last line should be `(contains-flatt? (rest alon))`:

```
; List-of-names -> Boolean
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
     (... (string=? (first alon) "Flatt") ...
      ... (contains-flatt? (rest alon)) ...)]))
```

The trick is now to combine the values of the two expressions in the appropriate manner. As mentioned, if the first expression yields `#true`, there is no need to search the rest of the list; if it is `#false`, though, the second expression may still yield `#true`, meaning the name `"Flatt"` is on the rest of the list. All of this suggests that the result of `(contains-flatt? alon)` is `#true` if **either** the first expression in the last line **or** the second expression yields `#true`.

```
; List-of-names -> Boolean
; determines whether "Flatt" occurs on alon
(check-expect
  (contains-flatt? (cons "X" (cons "Y"  (cons "Z" '()))))
  #false)
(check-expect
  (contains-flatt? (cons "A" (cons "Flatt" (cons "C" '()))))
  #true)
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
     (or (string=? (first alon) "Flatt")
         (contains-flatt? (rest alon)))]))
```

Figure 47: Searching a list

Figure 47 then shows the complete definition. Overall it doesn't look too different from the definitions in the first chapter of the book. It consists of a signature, a purpose statement, two examples, and a definition. The only way in which this function definition differs from anything you have seen before is the self-reference, that is, the reference to `contains-flatt?` in the body of the `define`. Then again, the data definition is self-referential, too, so in some sense the self-reference in the function shouldn't be too surprising.

Exercise 132. Use DrRacket to run `contains-flatt?` in this example:

```
(cons "Fagan"
  (cons "Findler"
    (cons "Fisler"
      (cons "Flanagan"
        (cons "Flatt"
          (cons "Felleisen"
            (cons "Friedman" '())))))))
```

What answer do you expect?

Exercise 133. Here is another way of formulating the second `cond` clause in `contains-flatt?`:

```
... (cond
      [(string=? (first alon) "Flatt") #true]
      [else (contains-flatt? (rest alon))]) ...
```

Explain why this expression produces the same answers as the `or` expression in the version of figure 47. Which version is better? Explain.

Exercise 134. Develop the `contains?` function, which determines whether some given string occurs on a given list of strings.

Note BSL actually comes with `member?`, a function that consumes two values and checks whether the first occurs in the second, a list:

```
> (member? "Flatt" (cons "b" (cons "Flatt" '())))
#true
```

Don't use `member?` to define the `contains?` function.

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
==
(cond
  [(empty? (cons "Flatt" (cons "C" '()))) #false]
  [(cons? (cons "Flatt" (cons "C" '())))
   (or
    (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
    (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
```

Figure 48: Computing with lists, step 1

8.4 Computing with Lists

Since we are still using BSL, the rules of algebra—see intermezzo 1—tell us how to determine the value of expressions such as

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
```

without DrRacket. Programmers must have an intuitive understanding of how this kind of calculation works, so we step through the one for this simple example.

Figure 48 displays the first step, which uses the usual substitution rule to determine the value of an application. The result is a conditional expression because, as an algebra teacher would say, the function is defined in a step-wise fashion.

```
...
==
(cond
 [#false #false]
 [(cons? (cons "Flatt" (cons "C" '())))
  (or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
      (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
==
(cond
 [(cons? (cons "Flatt" (cons "C" '())))
  (or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
      (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
==
(cond
 [#true
  (or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
      (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
==
(or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
    (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
```

Figure 49: Computing with lists, step 2

The calculation is continued in figure 49. To find the correct clause of the `cond` expression, we must determine the value of the conditions, one by one. Since a `cons`ed list isn't empty, the first condition's result is `#false`, and we therefore eliminate the first `cond` clause. Finally the condition in the second clause evaluates to `#true` because `cons?` of a `cons`ed list holds.

```
...
==
(or (string=? "Flatt" "Flatt")
    (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
== (or #true (contains-flatt? ...))
== #true
```

Figure 50: Computing with lists, step 3

From here, it is just three more steps of arithmetic to get the final result. Figure 50 displays the three steps. The first evaluates `(first (cons "Flatt" ...))` to `"Flatt"` due to the laws for `first`. The second discovers that `"Flatt"` is a

string and equal to `"Flatt"`. The third says `(or #true X)` is `#true` regardless of what `X` is.

Exercise 135. Use DrRacket's stepper to check the calculation for

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
```

Also use the stepper to determine the value of

```
(contains-flatt?
  (cons "A" (cons "Flatt" (cons "C" '()))))
```

What happens when `"Flatt"` is replaced with `"B"`?

Exercise 136. Validate with DrRacket's stepper

```
(our-first (our-cons "a" '())) == "a"
(our-rest (our-cons "a" '())) == '()
```

See What Is `'()`, What Is `cons` for the definitions of these functions.

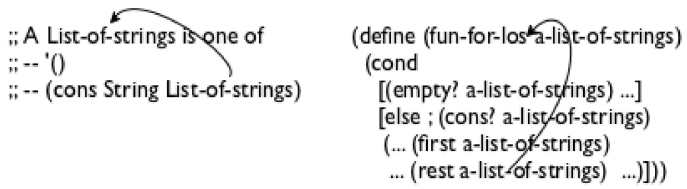


Figure 51: Arrows for self-references in data definitions and templates

9 Designing with Self-Referential Data Definitions

At first glance, self-referential data definitions seem to be far more complex than those for mixed data. But, as the example of `contains-flatt?` shows, the six steps of the design recipe still work. Nevertheless, in this section we generalize the design recipe so that it works even better for self-referential data definitions. The new parts concern the process of discovering when a self-referential data definition is needed, deriving a template, and defining the function body:

1. If a problem statement is about information of arbitrary size, you need a self-referential data definition to represent it. At this point, you have seen only one such class, List-of-names. The left side of figure 51 shows how to define *List-of-strings* in the same way. Other lists of atomic data work the same way.

   > Numbers also seem to be arbitrarily large. For inexact numbers, this is an illusion. For precise integers, this is indeed the case. Dealing with integers is therefore a part of this chapter.

   For a self-referential data definition to be valid, it must satisfy two conditions. First, it must contain at least two clauses. Second, at least one of the clauses must not refer back to the class of data that is being defined. It is good practice to identify the self-references explicitly with arrows from the references in the data definition back to the term being defined; see figure 51 for an example of such an annotation.

   You must check the validity of self-referential data definitions with the creation of data examples. Start with the clause that does not refer to the data definition; continue with the other one, using the first example where the clause refers to the definition itself. For the data definition in figure 51, you thus get lists like the following three:

   ```
   '()                by the first clause
   (cons "a" '())     by the second clause, previous example
   (cons "b"          by the second clause, previous example
     (cons "a"
       '()))
   ```

   If it is impossible to generate examples from the data definition, it is invalid. If you can generate examples but you can't see how to generate increasingly larger examples, the definition may not live up to its interpretation.

2. Nothing changes about the header material: the signature, the purpose statement, and the dummy definition. When you do formulate the purpose statement, focus on **what** the function computes **not how** it goes about it, especially not how it goes through instances of the given data.

   Here is an example to make this design recipe concrete:

   ```
   ; List-of-strings -> Number
   ; counts how many strings alos contains
   ```

```
(define (how-many alos)
  0)
```

The purpose statement clearly states that the function just counts the strings on the given input; there is no need to think ahead about how you might formulate this idea as a BSL function.

3. When it comes to functional examples, be sure to work through inputs that use the self-referential clause of the data definition several times. It is the best way to formulate tests that cover the entire function definition later.

   For our running example, the purpose statement almost generates functional examples by itself from the data examples:

given	wanted
`'()`	`0`
`(cons "a" '())`	`1`
`(cons "b" (cons "a" '()))`	`2`

   The first row is about the empty list, and we know that empty list contains nothing. The second row is a list of one string, so `1` is the desired answer. The last row is about a list of two strings.

4. At the core, a self-referential data definition looks like a data definition for mixed data. The development of the template can therefore proceed according to the recipe in Itemizations and Structures. Specifically, we formulate a `cond` expression with as many `cond` clauses as there are clauses in the data definition, match each recognizing condition to the corresponding clause in the data definition, and write down appropriate selector expressions in all `cond` lines that process compound values.

Question	**Answer**
Does the data definition distinguish among different sub-classes of data?	Your template needs as many `cond` clauses as sub-classes that the data definition distinguishes.
How do the sub-classes differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate “natural recursions” for the template to represent the self-references of the data definition.
If the data definition refers to some other data definition, where is this cross-reference to another data definition?	Specialize the template for the other data definition. Refer to this template. See Designing with Itemizations, Again, steps 4 and 5 of the design recipe.

Figure 52: How to translate a data definition into a template

Figure 52 expresses this idea as a question-and-answer game. In the left column it states questions about the data definition for the argument, and in the right column it explains what the answer means for the construction of the template.

If you ignore the last row and apply the first three questions to any function that consumes a List-of-strings, you arrive at this shape:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ... (rest alos) ...)]))
```

Recall, though, that the purpose of a template is to express the data definition as a function layout. That is, a template expresses as code what the data definition for the input expresses as a mix of English and BSL. Hence all important pieces of the data definition must find a counterpart in the template, and this guideline should also hold when a data definition is self-referential—contains an arrow from inside the definition to the term being defined. In particular, when a data definition is self-referential in the *i*th clause and the *k*th field of the structure mentioned there, the template should be self-referential in the *i*th `cond` clause and the selector expression for the *k*th field. For each such selector expression, add an arrow back to the function parameter. At the end, your template must have as many arrows as we have in the data definition.

Figure 51 illustrates this idea with the template for functions that consume List-of-strings shown side by side with the data definition. Both contain one arrow that originates in the second clause—the `rest` field and selector, respectively—and points back to the top of the respective definitions.

Since BSL and most programming languages are text-oriented, you must use an alternative to the arrow, namely, a self-application of the function to the appropriate selector expression:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
```

```
      (... (first alos) ...
       ... (fun-for-los (rest alos)) ...)]))
```

We refer to a self-use of a function as *recursion* and in the first four parts of the book as *natural recursion*.

5. For the function body we start with those `cond` lines without recursive function calls, known as *base cases*.The corresponding answers are typically easy to formulate or already given as examples.

   Then we deal with the self-referential cases. We start by reminding ourselves what each of the expressions in the template line computes. For the natural recursion we assume that the function already works as specified in our purpose statement. This last step is a leap of faith, but as you will see, it always works.

   > For the curious among our readers, the design recipe for arbitrarily large data corresponds to so-called "proofs by induction" in mathematics, and the "leap of faith" represents the use of the induction hypothesis for the inductive step of such a proof. Logic proves the validity of this proof technique with an Induction Theorem.

   The rest is then a matter of combining the various values.

Question	Answer
What are the answers for the non-recursive `cond` clauses?	The examples should tell you which values you need here. If not, formulate appropriate examples and tests.
What do the selector expressions in the recursive clauses compute?	The data definitions tell you what kind of data these expressions extract, and the interpretations of the data definitions tell you what this data represents.
What do the natural recursions compute?	Use the purpose statement of the function to determine what the value of the recursion means, **not how it computes this answer**. If the purpose statement doesn't tell you the answer, improve the purpose statement.
How can the function combine these values to get the desired answer?	Find a function in BSL that combines the values. Or, if that doesn't work, make a wish for a helper function. For many functions, this last step is straightforward. The purpose, the examples, and the template together tell you which function or expression **combines** the available values into the proper result. We refer to this function or expression as a *combinator*, slightly abusing existing terminology.

Figure 53: How to turn a template into a function definition

Question	Answer
So, if you are stuck here, ...	... arrange the examples from the third step in a table. Place the given input in the first column and the desired output in the last column. In the intermediate columns enter the values of the selector expressions and the natural recursion(s). Add examples until you see a pattern emerge that suggests a combinator.
If the template refers to some other template, what does the auxiliary function compute?	Consult the other function's purpose statement and examples to determine what it computes, and assume you may use the result even if you haven't finished the design of this helper function.

Figure 54: Turning a template into a function, the table method

Figure 53 formulates the first four questions and answers for this step. Let's use this game to complete the definition of `how-many`. Renaming the `fun-for-los` template to `how-many` gives us this much:

```
; List-of-strings -> Number
; determines how many strings are on alos
(define (how-many alos)
  (cond
    [(empty? alos) ...]
    [else
     (... (first alos) ...
      ... (how-many (rest alos)) ...)]))
```

As the functional examples already suggest, the answer for the base case is `0`. The two expressions in the second clause compute the `first` item and the number of strings in `(rest alos)`. To compute how many strings there are on all of `alos`, the function just needs to add `1` to the value of the latter expression:

```
(define (how-many alos)
  (cond
    [(empty? alos) 0]
    [else (+ (how-many (rest alos)) 1)]))
```

> Felix Klock suggested this table-based approach to guessing the combinator.

Finding the correct way to combine the values into the desired answer isn't always as easy. Novice programmers often get stuck with this step. As figure 54 suggests, it is a good idea to arrange the functional examples into a table that also spells out the values of the expressions in the template. Figure 55 shows what this table looks like for our `how-many` example. The left-most column lists the sample inputs, while the right-most column contains the desired answers for these inputs. The three columns in between show the values of the template expressions: `(first alos)`, `(rest alos)`, and `(how-many (rest alos))`, which is the natural recursion. If you stare at this table long enough, you recognize that the result column is always one more than the values in the natural recursion column. You may thus guess that

```
(+ (how-many (rest alos)) 1)
```

is the expression that computes the desired result. Since DrRacket is fast at checking these kinds of guesses, plug it in and click *RUN*. If the examples-turned-into-tests pass, think through the expression to convince yourself it works for all lists; otherwise add more example rows to the table until you have a different idea.

The table also points out that some selector expressions in the template are possibly irrelevant for the actual definition. Here `(first alos)` is not needed to compute the final answer—which is quite a contrast to `contains-flatt?`, which uses both expressions from the template.

As you work your way through the rest of this book, keep in mind that, in many cases, the combination step can be expressed with BSL's primitives, say, `+`, `and`, or `cons`. In some cases, though, you may have to make a wish, that is, design an auxiliary function. Finally, in yet other cases, you may need nested conditions.

6. Finally, make sure to turn all examples into tests, that these tests pass, and that running them covers all the pieces of the function.

   Here are our examples for `how-many` turned into tests:

   ```
   (check-expect (how-many '()) 0)
   (check-expect (how-many (cons "a" '())) 1)
   (check-expect
     (how-many (cons "b" (cons "a" '()))) 2)
   ```

   Remember, it is best to formulate examples directly as tests, and BSL allows this. Doing so also helps if you need to resort to the table-based guessing approach of the preceding step.

alos	(first alos)	(rest alos)	(how-many (rest alos))	(how-many alos)
(cons "a" '())	"a"	'()	0	1
(cons "b" (cons "a" '()))	"b"	(cons "a" '())	1	2
(cons "x" (cons "b" (cons "a" '())))	"x"	(cons "b" (cons "a" '()))	2	3

Figure 55: Tabulating arguments, intermediate values, and results

Figure 56 summarizes the design recipe of this section in a tabular format. The first column names the steps of the design recipe, and the second the expected results of each step. In the third column, we describe the activities that get you there. The figure is tailored to the kind of self-referential list definitions we use in this chapter. As always, practice helps you master the process, so we strongly recommend that you tackle the following exercises, which ask you to apply the recipe to several kinds of examples.

> You may want to copy figure 56 onto one side of an index card and write down your favorite versions of the questions and answers for this design recipe onto the back of it. Then carry it with you for future reference.

steps	outcome	activity
problem analysis	data definition	Develop a data representation for the information; create examples for specific items of information and interpret data as information; identify self-

		references.
header	signature; purpose; dummy definition	Write down a signature using defined names; formulate a concise purpose statement; create a dummy function that produces a constant value from the specified range.
examples	examples and tests	Work through several examples, at least one per clause in the data definition.
template	function template	Translate the data definition into a template: one `cond` clause per data clause; selectors where the condition identifies a structure; one natural recursion per self-reference.
definition	full-fledged definition	Find a function that combines the values of the expressions in the `cond` clauses into the expected answer.
test	validated tests	Turn them into `check-expect` tests and run them.

Figure 56: Designing a function for self-referential data

9.1 Finger Exercises: Lists

Exercise 137. Compare the template for `contains-flatt?` with the one for `how-many`. Ignoring the function name, they are the same. Explain the similarity.

Exercise 138. Here is a data definition for representing sequences of amounts of money:

```
; A List-of-amounts is one of:
; – '()
; – (cons PositiveNumber List-of-amounts)
```

Create some examples to make sure you understand the data definition. Also add an arrow for the self-reference.

Design the `sum` function, which consumes a List-of-amounts and computes the sum of the amounts. Use DrRacket's stepper to see how `(sum l)` works for a short list `l` in List-of-amounts.

Exercise 139. Now take a look at this data definition:

```
; A List-of-numbers is one of:
; – '()
; – (cons Number List-of-numbers)
```

Some elements of this class of data are appropriate inputs for `sum` from exercise 138 and some aren't.

Design the function `pos?`, which consumes a List-of-numbers and determines whether all numbers are positive numbers. In other words, if `(pos? l)` yields `#true`, then `l` is an element of List-of-amounts. Use DrRacket's stepper to understand how `pos?` works for `(cons 5 '())` and `(cons -1 '())`.

Also design `checked-sum`. The function consumes a List-of-numbers. It produces their sum if the input also belongs to List-of-amounts; otherwise it signals an error. **Hint** Recall to use check-error.

What does `sum` compute for an element of List-of-numbers?

Exercise 140. Design the function `all-true`, which consumes a list of Boolean values and determines whether all of them are `#true`. In other words, if there is any `#false` on the list, the function produces `#false`.

Now design `one-true`, a function that consumes a list of Boolean values and determines whether at least one item on the list is `#true`.

Employ the table-based approach to coding. It may help with the base case. Use DrRacket's stepper to see how these functions process the lists `(cons #true '())`, `(cons #false '())`, and `(cons #true (cons #false '()))`.

Exercise 141. If you are asked to design the function `cat`, which consumes a list of strings and appends them all into one long string, you are guaranteed to end up with this partial definition:

```
; List-of-string -> String
; concatenates all strings in l into one long string

(check-expect (cat '()) "")
(check-expect (cat (cons "a" (cons "b" '()))) "ab")
(check-expect
  (cat (cons "ab" (cons "cd" (cons "ef" '()))))
  "abcdef")

(define (cat l)
```

```
  (cond
    [(empty? l) ""]
    [else (... (first l) ... (cat (rest l)) ...)]))
```

l	(first l)	(rest l)	(cat (rest l))	(cat l)
(cons "a" (cons "b" '()))	???	???	???	"ab"
(cons "ab" (cons "cd" (cons "ef" '())))	???	???	???	"abcdef"

Figure 57: A table for `cat`

Fill in the table in figure 57. Guess a function that can create the desired result from the values computed by the sub-expressions.

Use DrRacket's stepper to evaluate `(cat (cons "a" '()))`.

Exercise 142. Design the `ill-sized?` function, which consumes a list of images `loi` and a positive number `n`. It produces the first image on `loi` that is not an `n` by `n` square; if it cannot find such an image, it produces `#false`.

Hint Use

```
; ImageOrFalse is one of:
; – Image
; – #false
```

for the result part of the signature.

9.2 Non-empty Lists

Now you know enough to use `cons` and to create data definitions for lists. If you solved (some of) the exercises at the end of the preceding section, you can deal with lists of various flavors of numbers, lists of Boolean values, lists of images, and so on. In this section we continue to explore what lists are and how to process them.

Let's start with the simple-looking problem of computing the average of a list of temperatures. To simplify, we provide the data definitions:

```
; A List-of-temperatures is one of:
; – '()
; – (cons CTemperature List-of-temperatures)

; A CTemperature is a Number greater than -273.
```

For our intentions, you should think of temperatures as plain numbers, but the second data definition reminds you that in reality not all numbers are temperatures and you should keep this in mind.

The header material is straightforward:

```
; List-of-temperatures -> Number
; computes the average temperature
(define (average alot) 0)
```

Making up examples for this problem is also easy, and so we just formulate one test:

```
(check-expect
  (average (cons 1 (cons 2 (cons 3 '())))) 2)
```

The expected result is of course the sum of the temperatures divided by the number of temperatures.

A moment's thought tells you that the template for average should be similar to the ones we have seen so far:

```
(define (average alot)
  (cond
    [(empty? alot) ...]
    [(cons? alot)
     (... (first alot) ...
```

```
      ... (average (rest alot)) ...)]))
```

The two `cond` clauses mirror the two clauses of the data definition; the questions distinguish empty lists from non-empty lists; and the natural recursion is needed because of the self-reference in the data definition.

It is way too difficult, however, to turn this template into a function definition. The first `cond` clause needs a number that represents the average of an empty collection of temperatures, but there is no such number. Similarly, the second clause demands a function that combines a temperature and an average for the remaining temperatures into a new average. Although possible, computing the average in this way is highly unnatural.

When we compute the average of temperatures, we divide their sum by their number. We said so when we formulated our trivial little example. This sentence suggests that `average` is a function of three tasks: summing, counting, and dividing. Our guideline from Fixed-Size Data tells us to write one function per task, and if we do so the design of average is obvious:

```
; List-of-temperatures -> Number
; computes the average temperature
(define (average alot)
  (/ (sum alot) (how-many alot)))

; List-of-temperatures -> Number
; adds up the temperatures on the given list
(define (sum alot) 0)

; List-of-temperatures -> Number
; counts the temperatures on the given list
(define (how-many alot) 0)
```

The last two function definitions are wishes, of course, for which we need to design complete definitions. Doing so is easy because `how-many` from above works for List-of-strings and List-of-temperatures (why?) and because the design of `sum` follows the same old routine:

```
; List-of-temperatures -> Number
; adds up the temperatures on the given list
(define (sum alot)
  (cond
    [(empty? alot) 0]
    [else (+ (first alot) (sum (rest alot)))]))
```

Stop! Use the example for `average` to create one for `sum` and ensure that the test runs properly. Then run the tests for `average`.

When you read this definition of average now, it is obviously correct simply because it directly corresponds to what everyone learns about averaging in school. Still, programs run not just for us but for others. In particular, others should be able to read the signature and use the function and expect an informative answer. But, our definition of `average` does not work for empty lists of temperatures.

Exercise 143. Determine how `average` behaves in DrRacket when applied to the empty list. Then design `checked-average`, a function that produces an informative error message when it is applied to `'()`.

> In mathematics, we would say exercise 143 shows that `average` is a *partial function* because it raises an error for `'()`.

An alternative solution is to inform future readers through the signature that `average` doesn't work for empty lists. For that, we need a data representation for lists that excludes `'()`, something like this:

```
; An NEList-of-temperatures is one of:
; – ???
; – (cons CTemperature NEList-of-temperatures)
```

The question is with what we should replace "???" so that the `'()` list is excluded but all other lists of temperatures are still constructable. One hint is that while the empty list is the shortest list, any list of one temperature is the next shortest list. In turn, this suggests that the first clause should describe all possible lists of one temperature:

```
; An NEList-of-temperatures is one of:
; – (cons CTemperature '())
; – (cons CTemperature NEList-of-temperatures)
; interpretation non-empty lists of Celsius temperatures
```

While this definition differs from the preceding list definitions, it shares the critical elements: a self-reference and a clause that does **not** use a self-reference. Strict adherence to the design recipe demands that you make up some examples of

NEList-of-temperatures to ensure that the definition makes sense. As always, you should start with the base clause, meaning the example must look like this:

```
(cons c '())
```

where c stands for a CTemperature, like thus: (cons -273 '()). Also, it is clear that all non-empty elements of List-of-temperatures are also elements of the new class of data: (cons 1 (cons 2 (cons 3 '()))) fits the bill if (cons 2 (cons 3 '())) does, and (cons 2 (cons 3 '())) belongs to NEList-of-temperatures because (cons 3 '()) is an element of NEList-of-temperatures, as confirmed before. Check for yourself that there is no limit on the size of NEList-of-temperatures.

Let's now return to the problem of designing average so that everyone knows it is for non-empty lists only. With the definition of NEList-of-temperatures, we now have the means to say what we want in the signature:

> This alternative development explains that, in this case, we can narrow down the domain of average and create a *total function*.

```
; NEList-of-temperatures -> Number
; computes the average temperature

(check-expect (average (cons 1 (cons 2 (cons 3 '()))))
              2)

(define (average ne-l)
  (/ (sum ne-l)
     (how-many ne-l)))
```

Naturally the rest remains the same: the purpose statement, the example-test, and the function definition. After all, the very idea of computing the average assumes a non-empty collection of numbers, and that was the entire point of our discussion.

Exercise 144. Will sum and how-many work for NEList-of-temperatures even though they are designed for inputs from List-of-temperatures? If you think they don't work, provide counter-examples. If you think they would, explain why.

Nevertheless, the definition also raises the question how to design sum and how-many because they consume instances of NEList-of-temperatures now. Here is the obvious result of the first three steps of the design recipe:

```
; NEList-of-temperatures -> Number
; computes the sum of the given temperatures
(check-expect
  (sum (cons 1 (cons 2 (cons 3 '())))) 6)
(define (sum ne-l) 0)
```

The example is adapted from the example for average; the dummy definition produces a number, but the wrong one for the given test.

The fourth step is the most interesting part of the design of sum for NEList-of-temperatures. All preceding examples of design demand a template that distinguishes empty lists from non-empty, that is, consed, lists because the data definitions have an appropriate shape. This is not true for NEList-of-temperatures. Here both clauses add consed lists. The two clauses differ, however, in the rest field of these lists. In particular, the first clause always uses '() in the rest field and the second one uses cons instead. Hence the proper condition to distinguish the first kind of data from the second extracts the rest field and then uses empty?:

```
; NEList-of-temperatures -> Number
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) ...]
    [else ...]))
```

Here else replaces (cons? (rest ne-l)).

Next you should inspect both clauses and determine whether one or both of them deal with ne-l as if it were a structure. This is of course the case, which the unconditional use of rest on ne-l demonstrates. Put differently, add appropriate selector expressions to the two clauses:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (... (first ne-l) ...)]
    [else (... (first ne-l) ... (rest ne-l) ...)]))
```

Before you read on, explain why the first clause does not contain the selector expression `(rest ne-l)`.

The final question of the template design concerns self-references in the data definition. As you know, `NEList-of-temperatures` contains one, and therefore the template for `sum` demands one recursive use:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (... (first ne-l) ...)]
    [else
     (... (first ne-l) ... (sum (rest ne-l)) ...)]))
```

Specifically, `sum` is called on `(rest ne-l)` in the second clause because the data definition is self-referential at the analogous point.

For the fifth design step, let's understand how much we already have. Since the first `cond` clause looks significantly simpler than the second one with its recursive function call, you should start with that one. In this particular case, the condition says that `sum` is applied to a list with exactly one temperature, `(first ne-l)`. Clearly, this one temperature is the sum of all temperatures on the given list:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (first ne-l)]
    [else
     (... (first ne-l) ... (sum (rest ne-l)) ...)]))
```

The second clause says that the list consists of a temperature and at least one more; `(first ne-l)` extracts the first position and `(rest ne-l)` the remaining ones. Furthermore, the template suggests to use the result of `(sum (rest ne-l))`. But `sum` is the function that you are defining, and you can't possibly know **how** it uses `(rest ne-l)`. All you do know is what the purpose statement says, namely, that `sum` adds all the temperatures on the given list, which is `(rest ne-l)`. If this statement is true, then `(sum (rest ne-l))` adds up all but one of the numbers of `ne-l`. To get the total, the function just has to add the first temperature:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (first ne-l)]
    [else (+ (first ne-l) (sum (rest ne-l)))]))
```

If you now run the test for this function, you will see that the leap of faith is justified. Indeed, for reasons beyond the scope of this book, this leap is **always** justified, which is why it is an inherent part of the design recipe.

Exercise 145. Design the `sorted>?` predicate, which consumes a NEList-of-temperatures and produces `#true` if the temperatures are sorted in descending order. That is, if the second is smaller than the first, the third smaller than the second, and so on. Otherwise it produces `#false`.

Hint This problem is another one where the table-based method for guessing the combinator works well. Here is a partial table for a number of examples in figure 58. Fill in the rest of the table. Then try to create an expression that computes the result from the pieces.

l	(first l)	(rest l)	(sorted>? (rest l))	(sorted>? l)
(cons 1 (cons 2 '()))	1	???	#true	#false
(cons 3 (cons 2 '()))	3	(cons 2 '())	???	#true
(cons 0 (cons 3 (cons 2 '())))	0	(cons 3 (cons 2 '()))	???	???

Figure 58: A table for `sorted>?`

Exercise 146. Design `how-many` for NEList-of-temperatures. Doing so completes `average`, so ensure that `average` passes all of its tests, too.

Exercise 147. Develop a data definition for *NEList-of-Booleans*, a representation of non-empty lists of Boolean values. Then redesign the functions `all-true` and `one-true` from exercise 140.

Exercise 148. Compare the function definitions from this section (`sum`, `how-many`, `all-true`, `one-true`) with the corresponding function definitions from the preceding sections. Is it better to work with data definitions that accommodate empty lists as opposed to definitions for non-empty lists? Why? Why not?

9.3 Natural Numbers

The BSL programming language supplies many functions that consume lists and a few that produce them, too. Among those is `make-list`, which consumes a number `n` together with some other value `v` and produces a list that contains `v` *n* times. Here are some examples:

```
> (make-list 2 "hello")
(cons "hello" (cons "hello" '()))
> (make-list 3 #true)
(cons #true (cons #true (cons #true '())))
> (make-list 0 17)
'()
```

In short, even though this function consumes atomic data, it produces arbitrarily large pieces of data. Your question should be how this is possible.

Our answer is that `make-list`'s input isn't just a number, it is a special kind of number. In kindergarten you called these numbers "counting numbers", that is, these numbers are used to count objects. In computer science, these numbers are dubbed *natural numbers*. Unlike regular numbers, natural numbers come with a data definition:

```
; An N is one of:
; – 0
; – (add1 N)
; interpretation represents the counting numbers
```

The first clause says that `0` is a natural number; it is used to say that there is no object to be counted. The second clause tells you that if *n* is a natural number, then $n+1$ is one too, because `add1` is a function that adds `1` to whatever number it is given. We could write this second clause as `(+ n 1)`, but the use of `add1` is supposed to signal that this addition is special.

What is special about this use of `add1` is that it acts more like a constructor from some structure-type definition than a regular function. For that reason, BSL also comes with the function `sub1`, which is the "selector" corresponding to `add1`. Given any natural number *m* not equal to `0`, you can use `sub1` to find out the number that went into the construction of *m*. Put differently, `add1` is like `cons` and `sub1` is like `first` and `rest`.

At this point you may wonder what the predicates are that distinguish `0` from those natural numbers that are not `0`. There are two, just as for lists: `zero?`, which determines whether some given number is `0`, and `positive?`, which determines whether some number is larger than `0`.

Now you are in a position to design functions on natural numbers, such as `make-list`, yourself. The data definition is already available, so let's add the header material:

```
; N String -> List-of-strings
; creates a list of n copies of s

(check-expect (copier 0 "hello") '())
(check-expect (copier 2 "hello")
              (cons "hello" (cons "hello" '())))

(define (copier n s)
  '())
```

Developing the template is the next step. The questions for the template suggest that `copier`'s body is a `cond` expression with two clauses: one for `0` and one for positive numbers. Furthermore, `0` is considered atomic and positive numbers are considered structured values, meaning the template needs a selector expression in the second clause. Last but not least, the data definition for `N` is self-referential in the second clause. Hence the template needs a recursive application to the corresponding selector expression in the second clause:

```
(define (copier n s)
  (cond
    [(zero? n) ...]
    [(positive? n) (... (copier (sub1 n) s) ...)]))
```

```
; N String -> List-of-strings
; creates a list of n copies of s
```

```
(check-expect (copier 0 "hello") '())
(check-expect (copier 2 "hello")
              (cons "hello" (cons "hello" '())))

(define (copier n s)
  (cond
    [(zero? n) '()]
    [(positive? n) (cons s (copier (sub1 n) s))]))
```

Figure 59: Creating a list of copies

Figure 59 contains a complete definition of the `copier` function, as obtained from its template. Let's reconstruct this step carefully. As always, we start with the `cond` clause that has no recursive calls. Here the condition tells us that the (important) input is `0`, and that means the function must produce a list with `0` items, that is, **none**. Of course, working through the second example has already clarified this case. Next we turn to the other `cond` clause and remind ourselves what its expressions compute:

1. `(sub1 n)` extracts the natural number that went into the construction of `n`, which we know is larger than `0`;

2. `(copier (sub1 n) s)` produces a list of `(sub1 n)` strings `s` according to the purpose statement of `copier`.

But the function is given `n` and must therefore produce a list with `n` strings `s`. Given a list with one too few strings, it is easy to see that the function must simply `cons` one `s` onto the result of `(copier (sub1 n) s)`. And that is precisely what the second clause specifies.

At this point, you should run the tests to ensure that this function works at least for the two worked examples. In addition, you may wish to use the function on some additional inputs.

Exercise 149. Does `copier` function properly when you apply it to a natural number and a Boolean or an image? Or do you have to design another function? Read Abstraction for an answer.

An alternative definition of `copier` might use `else`:

```
(define (copier.v2 n s)
  (cond
    [(zero? n) '()]
    [else (cons s (copier.v2 (sub1 n) s))]))
```

How do `copier` and `copier.v2` behave when you apply them to `0.1` and `"x"`? Explain. Use DrRacket's stepper to confirm your explanation.

Exercise 150. Design the function `add-to-pi`. It consumes a natural number `n` and adds it to `pi` **without** using the primitive `+` operation. Here is a start:

```
; N -> Number
; computes (+ n pi) without using +

(check-within (add-to-pi 3) (+ 3 pi) 0.001)

(define (add-to-pi n)
  pi)
```

Once you have a complete definition, generalize the function to `add`, which adds a natural number `n` to some arbitrary number `x` without using `+`. Why does the skeleton use `check-within`?

Exercise 151. Design the function `multiply`. It consumes a natural number `n` and multiplies it with a number `x` without using `*`.

Use DrRacket's stepper to evaluate `(multiply 3 x)` for any `x` you like. How does `multiply` relate to what you know from grade school?

Exercise 152. Design two functions: `col` and `row`.

The function `col` consumes a natural number `n` and an image `img`. It produces a column—a vertical arrangement—of `n` copies of `img`.

The function `row` consumes a natural number `n` and an image `img`. It produces a row—a horizontal arrangement—of `n` copies of `img`.

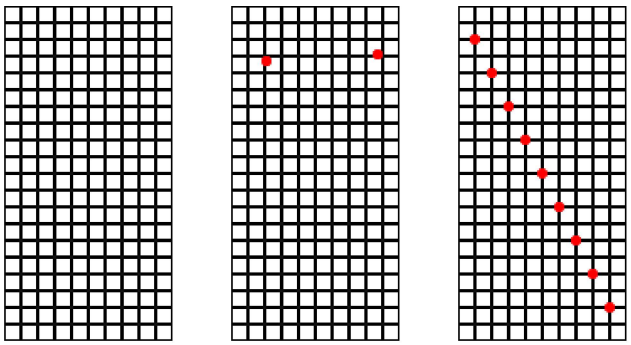


Figure 60: Random attacks

Exercise 153. The goal of this exercise is to visualize the result of a 1968-style European student riot. Here is the rough idea. A small group of students meets to make paint-filled balloons, enters some lecture hall, and randomly throws the balloons at the attendees. Your program displays how the balloons color the seats in the lecture hall.

Use the two functions from exercise 152 to create a rectangle of 8 by 18 squares, each of which has size 10 by 10. Place it in an `empty-scene` of the same size. This image is your lecture hall.

Design `add-balloons`. The function consumes a list of Posn whose coordinates fit into the dimensions of the lecture hall. It produces an image of the lecture hall with red dots added as specified by the Posns.

Figure 60 shows the output of **our** solution when given some list of Posns. The left-most is the clean lecture hall, the second one is after two balloons have hit, and the last one is a highly unlikely distribution of 10 hits. Where is the 10th?

9.4 Russian Dolls

Wikipedia defines a Russian doll as "a set of dolls of decreasing sizes placed one inside the other" and illustrates it with this picture:

In this picture, the dolls are taken apart so that the viewer can see them all.

> The problem may strike you as abstract or even absurd; it isn't clear why you would want to represent Russian dolls or what you would do with such a representation. Just play along for now.

Now consider the problem of representing such Russian dolls with BSL data. With a little bit of imagination, it is easy to see that an artist can create a nest of Russian dolls that consists of an arbitrary number of dolls. After all, it is always possible to wrap another layer around some given Russian doll. Then again, you also know that deep inside there is a solid doll without anything inside.

For each layer of a Russian doll, we could care about many different things: its size, though it is related to the nesting level; its color; the image that is painted on the surface; and so on. Here we just pick one, namely the color of the doll, which we represent with a string. Given that, we know that each layer of the Russian doll has two properties: its color and the doll that is inside. To represent pieces of information with two properties, we always define a structure type:

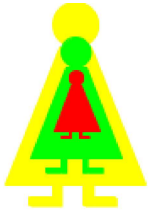
```
(define-struct layer [color doll])
```

And then we add a data definition:

```
; An RD (short for Russian doll) is one of:
; – String
; – (make-layer String RD)
```

Naturally, the first clause of this data definition represents the innermost doll, or, to be precise, its color. The second clause is for adding a layer around some given Russian doll. We represent this with an instance of `layer`, which obviously contains the color of the doll and one other field: the doll that is nested immediately inside of this doll.

Take a look at this doll:



It consists of three dolls. The red one is the innermost one, the green one sits in the middle, and the yellow is the current outermost wrapper. To represent this doll with an element of RD, you start on either end. We proceed from the inside out. The red doll is easy to represent as an RD. Since nothing is inside and since it is red, the string `"red"` will do fine. For the second layer, we use

```
(make-layer "green" "red")
```

which says that a green (hollow) doll contains a red doll. Finally, to get the outside we just wrap another layer around this last doll:

```
(make-layer "yellow" (make-layer "green" "red"))
```

This process should give you a good idea of how to go from any set of colored Russian dolls to a data representation. But keep in mind that a programmer must also be able to do the converse, that is, go from a piece of data to concrete information. In this spirit, draw a schematic Russian doll for the following element of RD:

```
(make-layer "pink" (make-layer "black" "white"))
```

You might even try this in BSL.

Now that we have a data definition and understand how to represent actual dolls and how to interpret elements of RD as dolls, we are ready to design functions that consume RDs. Specifically, let's design the function that counts how many dolls a Russian doll set contains. This sentence is a fine purpose statement and determines the signature, too:

```
; RD -> Number
; how many dolls are part of an-rd
```

As for data examples, let's start with `(make-layer "yellow" (make-layer "green" "red"))`. The image above tells us that `3` is the expected answer because there are three dolls: the red one, the green one, and the yellow one. Just working through this one example also tells us that when the input is a representation of this doll

then the answer is `1`.

Step four demands the development of a template. Using the standard questions for this step produces this template:

```
; RD -> Number
; how many dolls are a part of an-rd
(define (depth an-rd)
  (cond
    [(string? an-rd) ...]
    [(layer? an-rd)
     (... (layer-color an-rd) ...
      ... (depth (layer-doll an-rd)) ...)]))
```

The number of `cond` clauses is determined by the number of clauses in the definition of RD. Each of the clauses specifically spells out what kind of data it is about, and that tells us to use the `string?` and `layer?` predicates. While strings aren't compound data, instances of `layer` contain two values. If the function needs these values, it uses the selector expressions `(layer-color an-rd)` and `(layer-doll an-rd)`. Finally, the second clause of the data definition contains a self-reference from the *doll* field of the `layer` structure to the definition itself. Hence we need a recursive function call for the second selector expression.

The examples and the template almost dictate the function definition. For the non-recursive `cond` clause, the answer is obviously `1`. For the recursive clause, the template expressions compute the following results:

- `(layer-color an-rd)` extracts the string that describes the color of the current layer;
- `(layer-doll an-rd)` extracts the doll contained within the current layer; and
- `(depth (layer-doll an-rd))` determines how many dolls are part of `(layer-doll an-rd)`, according to the purpose statement of `depth`.

This last number is almost the desired answer but not quite because the difference between `an-rd` and `(layer-doll an-rd)` is one layer, meaning one extra doll. Put differently, the function must add `1` to the recursive result to obtain the actual answer:

```
; RD -> Number
; how many dolls are a part of an-rd
(define (depth an-rd)
  (cond
    [(string? an-rd) 1]
    [else (+ (depth (layer-doll an-rd)) 1)]))
```

Note how the function definition does not use `(layer-color an-rd)` in the second clause. Once again, we see that the template is an organization schema for everything we know about the data definition, but we may not need all of these pieces for the actual definition.

Let's finally translate the examples into tests:

```
(check-expect (depth "red") 1)
(check-expect
  (depth
   (make-layer "yellow" (make-layer "green" "red")))
  3)
```

If you run these in DrRacket, you will see that their evaluation touches all pieces of the definition of `depth`.

Exercise 154. Design the function `colors`. It consumes a Russian doll and produces a string of all colors, separated by a comma and a space. Thus our example should produce

```
"yellow, green, red"
```

Exercise 155. Design the function `inner`, which consumes an RD and produces the (color of the) innermost doll. Use DrRacket's stepper to evaluate `(inner rd)` for your favorite `rd`.

9.5 Lists and World

With lists and self-referential data definitions in general, you can design and run many more interesting world programs than with finite data. Just imagine you can now create a version of the space invader program from Itemizations and Structures that allows the player to fire as many shots from the tank as desired. Let's start with a simplistic version of this problem:

> If you haven't designed a world program in a while, reread Designing World Programs.

> **Sample Problem** Design a world program that simulates firing shots. Every time the "player" hits the space bar, the program adds a shot to the bottom of the canvas. These shots rise vertically at the rate of one pixel per tick.

Designing a world program starts with a separation of information into constants and elements of the ever-changing state of the world. For the former we introduce physical and graphical constants; for the latter we need to develop a data representation for world states. While the sample problem is relatively vague about the specifics, it clearly assumes a rectangular scenery with shots painted along a vertical line. Obviously the locations of the shots change with every clock tick, but the size of the scenery and x-coordinate of the line of shots remain the same:

```
(define HEIGHT 80) ; distances in terms of pixels
(define WIDTH 100)
(define XSHOTS (/ WIDTH 2))

; graphical constants
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define SHOT (triangle 3 "solid" "red"))
```

Nothing in the problem statement demands these particular choices, but as long as they are easy to change—meaning changing by editing a single definition—we have achieved our goal.

As for those aspects of the "world" that change, the problem statement mentions two. First, hitting the space bar adds a shot. Second, all the shots move straight up by one pixel per clock tick. Given that we cannot predict how many shots the player will "fire," we use a list to represent them:

```
; A List-of-shots is one of:
; – '()
; – (cons Shot List-of-shots)
```

```
; interpretation the collection of shots fired
```

The one remaining question is how to represent each individual shot. We already know that all of them have the same x-coordinate and that this coordinate stays the same throughout. Furthermore, all shots look alike. Hence, their y-coordinates are the only property in which they differ from each other. It therefore suffices to represent each shot as a number:

```
; A Shot is a Number.
; interpretation represents the shot's y-coordinate
```

We could restrict the representation of shots to the interval of numbers below `HEIGHT` because we know that all shots are launched from the bottom of the canvas and that they then move up, meaning their y-coordinate continuously decreases.

You can also use a data definition like this to represent this world:

```
; A ShotWorld is List-of-numbers.
; interpretation each number on such a list
;   represents the y-coordinate of a shot
```

After all, the above two definitions describe all list of numbers; we already have a definition for lists of numbers, and the name ShotWorld tells everyone what this class of data is about.

Once you have defined constants and developed a data representation for the states of the world, the key task is to pick which event handlers you wish to employ and to adapt their signatures to the given problem. The running example mentions clock ticks and the space bar, all of which translates into a wish list of three functions:

- the function that turns a world state into an image:

  ```
  ; ShotWorld -> Image
  ; adds the image of a shot for each  y on w
  ; at (MID,y} to the background image
  (define (to-image w) BACKGROUND)
  ```

  because the problem demands a visual rendering;

- one for dealing with tick events:

  ```
  ; ShotWorld -> ShotWorld
  ; moves each shot on w up by one pixel
  (define (tock w) w)
  ```

- and one function for dealing with key events:

  ```
  ; ShotWorld KeyEvent -> ShotWorld
  ; adds a shot to the world
  ; if the player presses the space bar
  (define (keyh w ke) w)
  ```

Don't forget that in addition to the initial wish list, you also need to define a `main` function that actually sets up the world and installs the handlers. Figure 61 includes this one function that is not designed but defined as a modification of standard schema.

Let's start with the design of `to-image`. We have its signature, purpose statement, and header, so we need examples next. Since the data definition has two clauses, there should be at least two examples: `'()` and a `cons`ed list, say, `(cons 9 '())`. The expected result for `'()` is obviously `BACKGROUND`; if there is a y-coordinate, though, the function must place the image of a shot at `MID` and the specified coordinate:

```
(check-expect (to-image (cons 9 '()))
              (place-image SHOT XSHOTS 9 BACKGROUND))
```

Before you read on, work through an example that applies `to-image` to a list of two Shots. Doing so helps understand **how** the function works.

The fourth step is about translating the data definition into a template:

```
; ShotWorld -> Image
(define (to-image w)
  (cond
    [(empty? w) ...]
    [else
     (... (first w) ... (to-image (rest w)) ...)]))
```

The template for data definitions for lists is so familiar now that it doesn't need much explanation. If you have any doubts, read over the questions in figure 52 and design the template on your own.

From here it is straightforward to define the function. The key is to combine the examples with the template and to answer the questions from figure 53. Following those, you start with the base case of an empty list of shots, and, from the examples, you know that the expected answer is `BACKGROUND`. Next you formulate what the template expressions in the second `cond` compute:

- `(first w)` extracts the first coordinate from the list;
- `(rest w)` is the rest of the coordinates; and
- `(to-image (rest w))` adds each shot on `(rest w)` to the background image, according to the purpose statement of `to-image`.

In other words, `(to-image (rest w))` renders the rest of the list as an image and thus performs almost all the work. What is missing is the first shot, `(first w)`. If you now apply the purpose statement to these two expressions, you get the desired expression for the second `cond` clause:

```
(place-image SHOT XSHOTS (first w)
             (to-image (rest w)))
```

The added icon is the standard image for a shot; the two coordinates are spelled out in the purpose statement; and the last argument to `place-image` is the image constructed from the rest of the list.

Figure 61 displays the complete function definition for `to-image` and indeed the rest of the program, too. The design of `tock` is just like the design of `to-image`, and you should work through it for yourself. The signature of the `keyh` handler, though, poses one interesting question. It specifies that the handler consumes two inputs with nontrivial data definitions. On the one hand, the ShotWorld is a self-referential data definition. On the other hand, the definition for KeyEvents is a large enumeration. For now, we have you "guess" which of the two arguments should drive the development of the template; later we will study such cases in depth.

```
; ShotWorld -> ShotWorld
(define (main w0)
  (big-bang w0
    [on-tick tock]
    [on-key keyh]
    [to-draw to-image]))

; ShotWorld -> ShotWorld
; moves each shot up by one pixel
(define (tock w)
  (cond
    [(empty? w) '()]
    [else (cons (sub1 (first w)) (tock (rest w)))]))

; ShotWorld KeyEvent -> ShotWorld
; adds a shot to the world if the space bar is hit
(define (keyh w ke)
  (if (key=? ke " ") (cons HEIGHT w) w))

; ShotWorld -> Image
; adds each shot y on w at (XSHOTS,y} to BACKGROUND
(define (to-image w)
  (cond
    [(empty? w) BACKGROUND]
    [else (place-image SHOT XSHOTS (first w)
                       (to-image (rest w)))]))
```

Figure 61: A list-based world program

As far as a world program is concerned, a key handler such as `keyh` is about the key event that it consumes. Hence, we consider it the main argument and use its data definition to derive the template. Specifically, following the data definition for KeyEvent from Enumerations, it dictates that the function needs a `cond` expression with numerous clauses like this:

```
(define (keyh w ke)
  (cond
    [(key=? ke "left") ...]
    [(key=? ke "right") ...]
    ...
    [(key=? ke " ") ...]
    ...
    [(key=? ke "a") ...]
```

```
    ...
    [(key=? ke "z") ...]))
```

Of course, just like for functions that consume all possible BSL values, a key handler usually does not need to inspect all possible cases. For our running problem, you specifically know that the key handler reacts only to the space bar and all others are ignored. So it is natural to collapse all of the `cond` clauses into an `else` clause except for the clause for `" "`.

Exercise 156. Equip the program in figure 61 with tests and make sure it passes those. Explain what `main` does. Then run the program via `main`.

Exercise 157. Experiment to determine whether the arbitrary decisions concerning constants are easy to change. For example, determine whether changing a single constant definition achieves the desired outcome:

- change the height of the canvas to 220 pixels;
- change the width of the canvas to 30 pixels;
- change the *x* location of the line of shots to “somewhere to the left of the middle”;
- change the background to a green rectangle; and
- change the rendering of shots to a red elongated rectangle.

Also check whether it is possible to double the size of the shot without changing anything else or to change its color to black.

Exercise 158. If you run `main`, press the space bar (fire a shot), and wait for a goodly amount of time, the shot disappears from the canvas. When you shut down the world canvas, however, the result is a world that still contains this invisible shot.

Design an alternative `tock` function that doesn’t just move shots one pixel per clock tick but also eliminates those whose coordinates place them above the canvas. **Hint** You may wish to consider the design of an auxiliary function for the recursive `cond` clause.

Exercise 159. Turn the solution of exercise 153 into a world program. Its main function, dubbed `riot`, consumes how many balloons the students want to throw; its visualization shows one balloon dropping after another at a rate of one per second. The function produces the list of Posns where the balloons hit.

Hints (1) Here is one possible data representation:

```
(define-struct pair [balloon# lob])
; A Pair is a structure (make-pair N List-of-posns)
; A List-of-posns is one of:
; – '()
; – (cons Posn List-of-posns)
; interpretation (make-pair n lob) means n balloons
; must yet be thrown and added to lob
```

(2) A `big-bang` expression is really just an expression. It is legitimate to nest it within another expression.

(3) Recall that `random` creates random numbers.

9.6 A Note on Lists and Sets

This book relies on your intuitive understanding of *sets* as collections of BSL values. The Universe of Data specifically says that a data definition introduces a name for a set of BSL values. There is one question that this book consistently asks about sets, and it is whether some element is in some given set. For example, `4` is in Number, while `"four"` is not. The book also shows how to use a data definition to check whether some value is a member of some named set and how to use some of the data definitions to generate sample elements of sets, but these two procedures are about data definitions, not sets per se.

At the same time, lists represent collections of values. Hence you might be wondering what the difference between a list and a set is or whether this is a needless distinction. If so, this section is for you.

Right now the primary difference between sets and lists is that the former is a concept we use to discuss steps in the design of code and the latter is one of many forms of data in BSL, our chosen programming language. The two ideas live at rather different levels in our conversations. However, given that a data definition introduces a data representation of actual information inside of BSL and given that sets are collections of information, you may now ask yourself how sets are represented inside of BSL as data.

> Most full-fledged languages directly support data representations of both lists and sets.

While lists have a special status in BSL, sets don't, but at the same time sets somewhat resemble lists. The key difference is the kind of functions a program normally uses with either form of data. BSL provides several basic constants and functions for lists—say, `empty`, `empty?`, `cons`, `cons?`, `first`, `rest`—and some functions that you could define yourself—for example, `member?`, `length`, `remove`, `reverse`, and so on. Here is an example of a function you can define but does not come with BSL

```
; List-of-string String -> N
; determines how often s occurs in los
(define (count los s)
  0)
```

Stop! Finish the design of this function.

Let's proceed in a straightforward and possibly naive manner and say sets are basically lists. And, to simplify further, let's focus on lists of numbers in this section. If we now accept that it merely matters whether a number is a part of a set or not, it is almost immediately clear that we can use lists in **two** different ways to represent sets.

```
; A Son.L is one of:          ; A Son.R is one of:
; – empty                     ; – empty
; – (cons Number Son.L)       ; – (cons Number Son.R)
;                             ;
; Son is used when it         ; Constraint If s is a Son.R,
; applies to Son.L and Son.R  ; no number occurs twice in s
```

Figure 62: Two data representations for sets

Figure 62 displays the two data definitions. Both basically say that a set is represented as a list of numbers. The difference is that the definition on the right comes with the constraint that no number may occur more than once on the list. After all, the key question we ask about a set is whether some number is in the set or not, and whether it is in a set once, twice, or three times makes no difference.

Regardless of which definition you choose, you can already define two important notions:

```
; Son
(define es '())

; Number Son -> Boolean
; is x in s
(define (in? x s)
  (member? x s))
```

The first one is the **empty set**, which in both cases is represented by the empty list. The second one is a membership test.

One way to build larger sets is to use `cons` and the above definitions. Say we wish to build a representation of the set that contains `1`, `2`, and `3`. Here is one such representation:

```
(cons 1 (cons 2 (cons 3 '())))
```

And it works for both data representations. But, is

```
(cons 2 (cons 1 (cons 3 '())))
```

really not a representation of the same set? Or how about

```
(cons 1 (cons 2 (cons 1 (cons 3 '()))))
```

The answer has to be affirmative as long as the primary concern is whether a number is in a set or not. Still, while the order of `cons` cannot matter, the constraint in the right-hand data definition rules out the last list as a Son.R because it contains `1` twice.

```
; Number Son.L -> Son.L
; removes x from s
(define s1.L
  (cons 1 (cons 1 '())))

(check-expect
  (set-.L 1 s1.L) es)

(define (set-.L x s)
  (remove-all x s))
```

```
; Number Son.R -> Son.R
; removes x from s
(define s1.R
  (cons 1 '()))

(check-expect
  (set-.R 1 s1.R) es)

(define (set-.R x s)
  (remove x s))
```

Figure 63: Functions for the two data representations of sets

The difference between the two data definitions shows up when we design functions. Say we want a function that removes a number from a set. Here is a wish-list entry that applies to both representations:

```
; Number Son -> Son
; subtracts x from s
(define (set- x s)
  s)
```

The purpose statement uses the word “subtract” because this is what logicians and mathematicians use when they work with sets.

Figure 63 shows the results. The two columns differ in two points:

1. The test on the left uses a list that contains `1` twice, while the one on the right represents the same set with a single `cons`.

2. Because of these differences, the `set-` on the left must use `remove-all`, while the one on the right gets away with `remove`.

Stop! Copy the code into the DrRacket definitions area and make sure the tests pass. Then read on and experiment with the code as you do.

An unappealing aspect of figure 63 is that the tests use `es`, a plain list, as the expected result. This problem may seem minor at first glance. Consider the following example, however:

```
(set- 1 set123)
```

where `set123` represents the set containing `1`, `2`, and `3` in one of two ways:

```
(define set123-version1
  (cons 1 (cons 2 (cons 3 '()))))

(define set123-version2
  (cons 1 (cons 3 (cons 2 '()))))
```

Regardless of which representation we choose, `(set- 1 set123)` evaluates to one of these two lists:

```
(define set23-version1
  (cons 2 (cons 3 '())))

(define set23-version2
  (cons 3 (cons 2 '())))
```

But we cannot predict which of those two `set-` produces.

For the simple case of two alternatives, it is possible to use the `check-member-of` testing facility as follows:

```
(check-member-of (set-.v1 1 set123.v1)
                 set23-version1
                 set23-version2)
```

If the expected set contains three elements, there are six possible variations, not including representations with repetitions, which the left-hand data definition allows.

Fixing this problem calls for the combination of two ideas. First, recall that `set-` is really about ensuring that the given element does not occur in the result. It is an idea that our way of turning the examples into tests does not bring across. Second, with BSL’s `check-satisfied` testing facility, it is possible to state precisely this idea.

Intermezzo 1: Beginning Student Language briefly mentions `check-satisfied`, but, in a nutshell, the facility determines whether an expression satisfies a certain property. A property is a function from values to Boolean. In our specific case, we wish to state that `1` is not a member of some set:

```
; Son -> Boolean
; #true if 1 a member of s;  #false otherwise
(define (not-member-1? s)
  (not (in? 1 s)))
```

Using `not-member-1?`, we can formulate the test case as follows:

```
(check-satisfied (set- 1 set123) not-member-1?)
```

and this variant clearly states what the function is supposed to accomplish. Better yet, this formulation simply does not depend on how the input or output set is represented.

In sum, lists and sets are related in that both are about collections of values, but they also differ strongly:

property	lists	sets
membership	one among many	critical
ordering	critical	irrelevant
# of occurrences	sensible	irrelevant
size	finite but arbitrary	finite or infinite

The last row in this table presents a new idea, though an obvious one, too. Many of the sets mentioned in this book are infinitely large, for example, Number, String, and also List-of-strings. In contrast, a list is **always** finite though it may contain an arbitrarily large number of items.

In sum, this section explains the essential differences between sets and lists and how to represent finite sets with finite lists in two different ways. BSL is not expressive enough to represent infinite sets; exercise 299 introduces a completely different representation of sets, a representation that can cope with infinite sets, too. The question of how actual programming languages represent sets is beyond the scope of this book, however.

Exercise 160. Design the functions `set+.L` and `set+.R`, which create a set by adding a number `x` to some given set `s` for the left-hand and right-hand data definition, respectively.

10 More on Lists

Lists are a versatile form of data that come with almost all languages now. Programmers have used them to build large applications, artificial intelligences, distributed systems, and more. This chapter illustrates some ideas from this world, including functions that create lists, data representations that call for structures inside of lists, and representing text files as lists.

10.1 Functions that Produce Lists

Here is a function for determining the wage of an hourly employee:

```
; Number -> Number
; computes the wage for h hours of work
(define (wage h)
  (* 12 h))
```

It consumes the number of hours worked and produces the wage. A company that wishes to use payroll software isn't interested in this function, however. It wants one that computes the wages for all its employees.

Call this new function `wage*`. Its task is to process all employee work hours and to determine the wages due to each of them. For simplicity, let's assume that the input is a list of numbers, each representing the number of hours that one employee worked, and that the expected result is a list of the weekly wages earned, also represented with a list of numbers.

Since we already have a data definition for the inputs and outputs, we can immediately move to the second design step:

```
; List-of-numbers -> List-of-numbers
; computes the weekly wages for the weekly hours
(define (wage* whrs)
  '())
```

Next you need some examples of inputs and the corresponding outputs. So you make up some short lists of numbers that represent weekly hours:

given	expected
`'()`	`'()`
`(cons 28 '())`	`(cons 336 '())`
`(cons 4 (cons 2 '()))`	`(cons 48 (cons 24 '()))`

In order to compute the output, you determine the weekly wage for each number on the given input list. For the first example, there are no numbers on the input list so the output is `'()`. Make sure you understand why the second and third expected outputs are what you want.

Given that `wage*` consumes the same kind of data as several other functions from Lists and given that a template depends only on the shape of the data definition, you can reuse this template:

```
(define (wage* whrs)
```

```
(cond
  [(empty? whrs) ...]
  [else (... (first whrs) ...
         ... (wage* (rest whrs)) ...)]))
```

In case you want to practice the development of templates, use the questions from figure 52.

It is now time for the most creative design step. Following the design recipe, we consider each `cond` line of the template in isolation. For the non-recursive case, `(empty? whrs)` is true, meaning the input is `'()`. The examples from above specify the desired answer, `'()`, and so we are done.

In the second case, the design questions tell us to state what each expression of the template computes:

- `(first whrs)` yields the first number on `whrs`, which is the first number of hours worked;
- `(rest whrs)` is the rest of the given list; and
- `(wage* (rest whrs))` says that the rest is processed by the very function we are defining. As always, we use its signature and its purpose statement to figure out the result of this expression. The signature tells us that it is a list of numbers, and the purpose statement explains that this list represents the list of wages for its input, which is the rest of the list of hours.

The key is to rely on these facts when you formulate the expression that computes the result in this case, even if the function is not yet defined.

Since we already have the list of wages for all but the first item of `whrs`, the function must perform two computations to produce the expected output for the **entire** `whrs`: compute the weekly wage for `(first whrs)` and construct the list that represents all weekly wages for `whrs`. For the first part, we reuse `wage`. For the second, we `cons` the two pieces of information together into one list:

```
(cons (wage (first whrs)) (wage* (rest whrs)))
```

And with that, the definition is complete: see figure 64.

```
; List-of-numbers -> List-of-numbers
; computes the weekly wages for all given weekly hours
(define (wage* whrs)
  (cond
    [(empty? whrs) '()]
    [else (cons (wage (first whrs)) (wage* (rest whrs)))]))

; Number -> Number
; computes the wage for h hours of work
(define (wage h)
  (* 12 h))
```

Figure 64: Computing the wages of all employees

Exercise 161. Translate the examples into tests and make sure they all succeed. Then change the function in figure 64 so that everyone gets $14 per hour. Now revise the entire program so that changing the wage for everyone is a single change to the **entire** program and not several.

Exercise 162. No employee could possibly work more than 100 hours per week. To protect the company against fraud, the function should check that no item of the input list of `wage*` exceeds 100. If one of them does, the function should immediately signal an error. How do we have to change the function in figure 64 if we want to perform this basic reality check?

Exercise 163. Design `convertFC`. The function converts a list of measurements in Fahrenheit to a list of Celsius measurements.

Exercise 164. Design the function `convert-euro`, which converts a list of US$ amounts into a list of € amounts. Look up the current exchange rate on the web.

> Show the products of the various steps in the design recipe. If you are stuck, show someone how far you got according to the design recipe. The recipe isn't just a design tool for you to use; it is also a diagnosis system so that others can help you help yourself.

Generalize `convert-euro` to the function `convert-euro*`, which consumes an exchange rate and a list of US$ amounts and converts the latter into a list of € amounts.

Exercise 165. Design the function `subst-robot`, which consumes a list of toy descriptions (one-word strings) and replaces all occurrences of `"robot"` with `"r2d2"`; all other descriptions remain the same.

Generalize `subst-robot` to `substitute`. The latter consumes two strings, called `new` and `old`, and a list of strings. It produces a new list of strings by substituting all occurrences of `old` with `new`.

10.2 Structures in Lists

Representing a work week as a number is a bad choice because the printing of a paycheck requires more information than hours worked per week. Also, not all employees earn the same amount per hour. Fortunately a list may contain items other than atomic values; indeed, lists may contain whatever values we want, especially structures.

Our running example calls for just such a data representation. Instead of numbers, we use structures that represent employees plus their work hours and pay rates:

```
(define-struct work [employee rate hours])
; A (piece of) Work is a structure:
;   (make-work String Number Number)
; interpretation (make-work n r h) combines the name
; with the pay rate r and the number of hours h
```

While this representation is still simplistic, it is just enough of an additional challenge because it forces us to formulate a data definition for lists that contain structures:

```
; Low (short for list of works) is one of:
; – '()
; – (cons Work Low)
; interpretation an instance of Low represents the
; hours worked for a number of employees
```

Here are three elements of Low:

```
'()
(cons (make-work "Robby" 11.95 39)
      '())
(cons (make-work "Matthew" 12.95 45)
      (cons (make-work "Robby" 11.95 39)
            '())))
```

Use the data definition to explain why these pieces of data belong to Low.

Stop! Also use the data definition to generate two more examples.

> When you work on real-world projects, you won't use such suffixes; instead you will use a tool for managing different versions of code.

Now that you know that the definition of Low makes sense, it is time to redesign the function `wage*` so that it consumes elements of Low, not just lists of numbers:

```
; Low -> List-of-numbers
; computes the weekly wages for the given records
(define (wage*.v2 an-low)
  '())
```

The suffix ".v2" at the end of the function name informs every reader of the code that this is a second, revised version of the function. In this case, the revision starts with a new signature and an adapted purpose statement. The header is the same as above.

The third step of the design recipe is to work through an example. Let's start with the second list above. It contains one work record, namely, `(make-work "Robby" 11.95 39)`. Its interpretation is that `"Robby"` worked for `39` hours and that he is paid at the rate of $11.95 per hour. Hence his wage for the week is $466.05, that is, `(* 11.95 39)`. The desired result for `wage*.v2` is therefore `(cons 466.05 '())`. Naturally, if the input list contained two work records, we would perform this kind of computation twice, and the result would be a list of two numbers. Stop! Determine the expected result for the third data example above.

Note on Numbers Keep in mind that BSL—unlike most other programming languages—understands decimal numbers just like you do, namely, as exact fractions. A language such as Java, for example, would produce 466.04999999999995 for the expected wage of the first work record. Since you cannot predict when operations on decimal numbers behave in this strange way, you are better off writing down such examples as

```
(check-expect
  (wage*.v2
```

```
    (cons (make-work "Robby" 11.95 39) '()))
  (cons (* 11.95 39) '()))
```

just to prepare yourself for other programming languages. Then again, writing down the example in this style also means you have really figured out how to compute the wage. **End**

From here we move on to the development of the template. If you use the template questions, you quickly get this much:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
     (... (first an-low) ...
      ... (wage*.v2 (rest an-low)) ...)]))
```

because the data definition consists of two clauses, because it introduces `'()` in the first clause and `cons`ed structures in the second, and so on. But you also realize that you know even more about the input than this template expresses. For example, you know that `(first an-low)` extracts a structure of three fields from the given list. This seems to suggest the addition of three more expressions to the template:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
     (... (first an-low) ...
      ... ... (work-employee (first an-low)) ...
      ... ... (work-rate (first an-low)) ...
      ... ... (work-hours (first an-low)) ...
      (wage*.v2 (rest an-low)) ...)]))
```

This template lists **all** potentially interesting data.

We use a different strategy here. Specifically, we suggest that you **create and refer to a separate function template** whenever you are developing a template for a data definition that refers to other data definitions:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
     (... (for-work (first an-low))
      ... (wage*.v2 (rest an-low)) ...)]))

; Work -> ???
; a template for processing elements of Work
(define (for-work w)
  (... (work-employee w) ...
   ... (work-rate w) ...
   ... (work-hours w) ...))
```

Splitting the templates leads to a natural partition of work into functions and among functions; none of them grows too large, and all of them relate to a specific data definition.

Finally, you are ready to program. As always you start with the simple-looking case, which is the first `cond` line here. If `wage*.v2` is applied to `'()`, you expect `'()` back and that settles it. Next you move on to the second line and remind yourself of what these expressions compute:

1. `(first an-low)` extracts the first `work` structure from the list;
2. `(for-work ...)` says that you wish to design a function that processes `work` structures;
3. `(rest an-low)` extracts the rest of the given list; and
4. `(wage*.v2 (rest an-low))` determines the list of wages for all the `work` records other than the first one, according to the purpose statement of the function.

If you are stuck here, use the table method from figure 54.

If you understand it all, you see that it is enough to `cons` the two expressions together:

```
... (cons (for-work (first an-low))
          (wage*.v2 (rest an-low))) ...
```

assuming that `for-work` computes the wage for the first work record. In short, you have finished the function by adding another entry to your wish list of functions.

Since `for-work` is a name that just serves as a stand-in and since it is a bad name for this function, let's call the function `wage.v2` and write down its complete wish-list entry:

```
; Work -> Number
; computes the wage for the given work record w
(define (wage.v2 w)
  0)
```

The design of this kind of function is extensively covered in Fixed-Size Data and thus doesn't need any additional explanation here. Figure 65 shows the final result of developing `wage` and `wage*.v2`.

```
; Low -> List-of-numbers
; computes the weekly wages for all weekly work records

(check-expect
  (wage*.v2 (cons (make-work "Robby" 11.95 39) '()))
  (cons (* 11.95 39) '()))

(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) '()]
    [(cons? an-low) (cons (wage.v2 (first an-low))
                          (wage*.v2 (rest an-low)))]))

; Work -> Number
; computes the wage for the given work record w
(define (wage.v2 w)
  (* (work-rate w) (work-hours w)))
```

Figure 65: Computing the wages from work records

Exercise 166. The `wage*.v2` function consumes a list of work records and produces a list of numbers. Of course, functions may also produce lists of structures.

Develop a data representation for paychecks. Assume that a paycheck contains two distinctive pieces of information: the employee's name and an amount. Then design the function `wage*.v3`. It consumes a list of work records and computes a list of paychecks from it, one per record.

In reality, a paycheck also contains an employee number. Develop a data representation for employee information and change the data definition for work records so that it uses employee information and not just a string for the employee's name. Also change your data representation of paychecks so that it contains an employee's name and number, too. Finally, design `wage*.v4`, a function that maps lists of revised work records to lists of revised paychecks.

Note on Iterative Refinement This exercise demonstrates the *iterative refinement* of a task. Instead of using data representations that include all relevant information, we started from simplistic representation of paychecks and gradually made the representation realistic. For this simple program, refinement is overkill; later we will encounter situations where iterative refinement is not just an option but a necessity.

Exercise 167. Design the function `sum`, which consumes a list of `Posn`s and produces the sum of all of its x-coordinates.

Exercise 168. Design the function `translate`. It consumes and produces lists of Posns. For each `(make-posn x y)` in the former, the latter contains `(make-posn x (+ y 1))`. We borrow the word "translate" from geometry, where the movement of a point by a constant distance along a straight line is called a *translation*.

Exercise 169. Design the function `legal`. Like `translate` from exercise 168, the function consumes and produces a list of Posns. The result contains all those Posns whose x-coordinates are between 0 and 100 and whose y-coordinates are between 0 and 200.

Exercise 170. Here is one way to represent a phone number:

```
(define-struct phone [area switch four])
; A Phone is a structure:
;   (make-phone Three Three Four)
; A Three is a Number between 100 and 999.
; A Four is a Number between 1000 and 9999.
```

Design the function `replace`. It consumes and produces a list of Phones. It replaces all occurrence of area code `713` with `281`.

Functions and Programs introduces `read-file`, a function for reading an entire text file as a string. In other words, the creator of `read-file` chose to represent text files as strings, and the function creates the data representation for specific files (specified by a name). Text files aren't plain long texts or strings, however. They are organized into lines and words, rows and cells, and many other ways. In short, representing the content of a file as a plain string might work on rare occasions but is usually a bad choice.

> Add `(require 2htdp/batch-io)` to your definitions area.

ttt.txt

```
                TTT

          Put up in a place
        where it's easy to see
       the cryptic admonishment
                T.T.T.

    When you feel how depressingly
          slowly you climb,
      it's well to remember that
          Things Take Time.

              Piet Hein
```

Figure 66: Things take time

For concreteness, take a look at the sample file in figure 66. It contains a poem by Piet Hein, and it consists of many lines and words. When you use the program

```
(read-file "ttt.txt")
```

to turn this file into a BSL string, you get this:

```
"TTT\n \nPut up in a place\nwhere ...."
```

> The dots aren't really a part of the result, as you probably guessed.

where the `"\n"` inside the string indicates line breaks.

While it is indeed possible to break apart this string with primitive operations on strings, for example, `explode`, most programming languages—including BSL—support many different representations of files and functions that create such representations from existing files:

- One way to represent this file is as a list of lines, where each line is represented as one string:

  ```
  (cons "TTT"
    (cons ""
      (cons "Put up in a place"
        (cons ...
          '()))))
  ```

  Here the second item of the list is the empty string because the file contains an empty line.

- Another way is to use a list of words, again each word represented as a string:

  ```
  (cons "TTT"
    (cons "Put"
      (cons "up"
        (cons "in"
          (cons ...
            '())))))
  ```

  Note how the empty second line disappears with this representation. After all, there are no words on the empty line.

- And a third representation relies on lists of lists of words:

  ```
  (cons (cons "TTT" '())
    (cons '()
      (cons (cons "Put"
              (cons "up"
                (cons ... '())))
        (cons ...
  ```

```
        '()))))
```

This representation has an advantage over the second one in that it preserves the organization of the file, including the emptiness of the second line. The price is that all of a sudden lists contain lists.

While the idea of list-containing lists may sound frightening at first, you need not worry. The design recipe helps even with such complications.

```
; String -> String
; produces the content of file f as a string
(define (read-file f) ...)

; String -> List-of-string
; produces the content of file f as a list of strings,
; one per line
(define (read-lines f) ...)

; String -> List-of-string
; produces the content of file f as a list of strings,
; one per word
(define (read-words f) ...)

; String -> List-of-list-of-string
; produces the content of file f as a list of list of
; strings, one list per line and one string per word
(define (read-words/line f) ...)

; The above functions consume the name of a file as a String
; argument. If the specified file does not exist in the
; same folder as the program, they signal an error.
```

Figure 67: Reading files

Before we get started, take a look at figure 67. It introduces a number of useful file reading functions. They are not comprehensive: there are many other ways of dealing with text from files, and you will need to know a lot more to deal with all possible text files. For our purposes here—teaching and learning the principles of systematic program design—they suffice, and they empower you to design reasonably interesting programs.

Figure 67 uses the names of two data definitions that do not exist yet, including one involving list-containing lists. As always, we start with a data definition, but this time we leave this task to you. Hence, before you read on, solve the following exercises. The solutions are needed to make complete sense out of the figure, and without working through the solutions, you cannot really understand the rest of this section.

Exercise 171. You know what the data definition for List-of-strings looks like. Spell it out. Make sure that you can represent Piet Hein's poem as an instance of the definition where each line is represented as a string and another instance where each word is a string. Use `read-lines` and `read-words` to confirm your representation choices.

Next develop the data definition for *List-of-list-of-strings*. Again, represent Piet Hein's poem as an instance of the definition where each line is represented as a list of strings, one per word, and the entire poem is a list of such line representations. You may use `read-words/line` to confirm your choice.

As you probably know, operating systems come with programs that measure files. One counts the number of lines, another determines how many words appear per line. Let us start with the latter to illustrate how the design recipe helps with the design of complex functions.

The first step is to ensure that we have all the necessary data definitions. If you solved the above exercise, you have a data definition for all possible inputs of the desired function, and the preceding section defines List-of-numbers, which describes all possible inputs. To keep things short, we use LLS to refer to the class of lists of lists of strings, and use it to write down the header material for the desired function:

```
; LLS -> List-of-numbers
; determines the number of words on each line
(define (words-on-line lls) '())
```

We name the functions `words-on-line` because it is appropriate and captures the purpose statement in one phrase.

What is really needed, though, is a set of **data examples**:

```
(define line0 (cons "hello" (cons "world" '())))
(define line1 '())
```

```
(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))
```

The first two definitions introduce two examples of lines: one contains two words, the other contains none. The last two definitions show how to construct instances of LLS from these line examples. Determine what the expected result is when the function is given these two examples.

Once you have data examples, it is easy to formulate functional examples; just imagine applying the function to each of the data examples. When you apply `words-on-line` to `lls0`, you should get the empty list back because there are no lines. When you apply `words-on-line` to `lls1`, you should get a list of two numbers back because there are two lines. The two numbers are `2` and `0`, respectively, given that the two lines in `lls1` contain two and no words each.

Here is how you translate all this into test cases:

```
(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1)
              (cons 2 (cons 0 '())))
```

By doing it at the end of the second step, you have a complete program, though running it just fails some of the test cases.

The development of the template is the interesting step for this sample problem. By answering the template questions from figure 52, you get the usual list-processing template immediately:

```
(define (words-on-line lls)
  (cond
    [(empty? lls) ...]
    [else
     (... (first lls) ; a list of strings
      ... (words-on-line (rest lls)) ...)]))
```

As in the preceding section, we know that the expression `(first lls)` extracts a List-of-strings, which has a complex organization, too. The temptation is to insert a nested template to express this knowledge, but as you should recall, the better idea is to develop a second auxiliary template and to change the first line in the second condition so that it refers to this auxiliary template.

Since this auxiliary template is for a function that consumes a list, the template looks nearly identical to the previous one:

```
(define (line-processor ln)
  (cond
    [(empty? lls) ...]
    [else
     (... (first ln) ; a string
      ... (line-processor (rest ln)) ...)]))
```

The important differences are that `(first ln)` extracts a string from the list, and we consider strings as atomic values. With this template in hand, we can change the first line of the second case in `words-on-line` to

```
... (line-processor (first lls)) ...
```

which reminds us for the fifth step that the definition for `words-on-line` may demand the design of an auxiliary function.

Now it is time to program. As always, we use the questions from figure 53 to guide this step. The first case, concerning empty lists of lines, is the easy case. Our examples tell us that the answer in this case is `'()`, that is, the empty list of numbers. The second case, concerning `cons`, contains several expressions, and we start with a reminder of what they compute:

- `(first lls)` extracts the first line from the non-empty list of (represented) lines;
- `(line-processor (first lls))` suggests that we may wish to design an auxiliary function to process this line;
- `(rest lls)` is the rest of the list of line; and
- `(words-on-line (rest lls))` computes a list of words per line for the rest of the list. How do we know this? We promised just that with the signature and the purpose statement for `words-on-line`.

Assuming we can design an auxiliary function that consumes a line and counts the words on one line—let's call it `words#`—it is easy to complete the second condition:

```
(cons (words# (first lls))
      (words-on-line (rest lls)))
```

This expression `cons`es the number of words on the first line of `lls` onto a list of numbers that represents the number of words on the remainder of the lines of `lls`.

It remains to design the `words#` function. Its template is dubbed `line-processor` and its purpose is to count the number of words on a line, which is just a list of strings. So here is the wish-list entry:

```
; List-of-strings -> Number
; counts the number of words on los
(define (words# los) 0)
```

At this point, you may recall the example used to illustrate the design recipe for self-referential data in Designing with Self-Referential Data Definitions. The function is called `how-many`, and it too counts the number of strings on a list of strings. Even though the input for `how-many` is supposed to represent a list of names, this difference simply doesn't matter; as long as it correctly counts the number of strings on a list of strings, `how-many` solves our problem.

Since it is good to reuse existing functions, you may define `words#` as

```
(define (words# los)
  (how-many los))
```

In reality, however, programming languages come with functions that solve such problems already. BSL calls this function `length`, and it counts the number of values on any list of values, no matter what the values are.

```
; An LLS is one of:
; – '()
; – (cons Los LLS)
; interpretation a list of lines, each is a list of Strings

(define line0 (cons "hello" (cons "world" '())))
(define line1 '())

(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))

; LLS -> List-of-numbers
; determines the number of words on each line

(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1) (cons 2 (cons 0 '())))

(define (words-on-line lls)
  (cond
    [(empty? lls) '()]
    [else (cons (length (first lls))
                (words-on-line (rest lls)))]))
```

Figure 68: Counting the words on a line

Figure 68 summarizes the full design for our sample problem. The figure includes two test cases. Also, instead of using the separate function `words#`, the definition of `words-on-line` simply calls the `length` function that comes with BSL. Experiment with the definition in DrRacket and make sure that the two test cases cover the entire function definition.

> You may wish to look over the list of functions that come with BSL. Some may look obscure but may become useful in one of the upcoming problems. Using such functions saves your time, not ours.

With one small step, you can now design your first file utility:

```
; String -> List-of-numbers
; counts the words on each line in the given file
(define (file-statistic file-name)
  (words-on-line
    (read-words/line file-name)))
```

It merely composes a library function with `words-on-line`. The former reads a file as a List-of-list-of-strings and hands this value to the latter.

This idea of composing a built-in function with a newly designed function is common. Naturally, people don't design functions randomly and expect to find something in the chosen programming language to complement their design. Instead, program designers plan ahead and design the function **to the output** that available functions deliver. More generally still and as mentioned above, it is common to think about a solution as a composition of two computations and to develop an appropriate data collection with which to communicate the result of one computation to the second one, where each computation is implemented with a function.

```
; 1String -> String
; converts the given 1String to a 3-letter numeric String

(check-expect (encode-letter "z") (code1 "z"))
(check-expect (encode-letter "\t")
              (string-append "00" (code1 "\t")))
(check-expect (encode-letter "a")
              (string-append "0" (code1 "a")))

(define (encode-letter s)
  (cond
    [(>= (string->int s) 100) (code1 s)]
    [(< (string->int s) 10)
     (string-append "00" (code1 s))]
    [(< (string->int s) 100)
     (string-append "0" (code1 s))]))

; 1String -> String
; converts the given 1String into a String

(check-expect (code1 "z") "122")

(define (code1 c)
  (number->string (string->int c)))
```

Figure 69: Encoding strings

Exercise 172. Design the function `collapse`, which converts a list of lines into a string. The strings should be separated by blank spaces (`" "`). The lines should be separated with a newline (`"\n"`).

Challenge When you are finished, use the program like this:

```
(write-file "ttt.dat"
            (collapse (read-words/line "ttt.txt")))
```

To make sure the two files `"ttt.dat"` and `"ttt.txt"` are identical, remove all extraneous white spaces in your version of the `T.T.T.` poem.

Exercise 173. Design a program that removes all articles from a text file. The program consumes the name `n` of a file, reads the file, removes the articles, and writes the result out to a file whose name is the result of concatenating `"no-articles-"` with `n`. For this exercise, an article is one of the following three words: `"a"`, `"an"`, and `"the"`.

Use `read-words/line` so that the transformation retains the organization of the original text into lines and words. When the program is designed, run it on the Piet Hein poem.

Exercise 174. Design a program that encodes text files numerically. Each letter in a word should be encoded as a numeric three-letter string with a value between 0 and 256. Figure 69 shows our encoding function for single letters. Before you start, explain these functions.

Hints (1) Use `read-words/line` to preserve the organization of the file into lines and words. (2) Read up on `explode` again.

Exercise 175. Design a BSL program that simulates the Unix command `wc`. The purpose of the command is to count the number of 1Strings, words, and lines in a given file. That is, the command consumes the name of a file and produces a value that consists of three numbers.

```
; Matrix -> Matrix
; transposes the given matrix along the diagonal

(define wor1 (cons 11 (cons 21 '())))
(define wor2 (cons 12 (cons 22 '())))
(define tam1 (cons wor1 (cons wor2 '())))

(check-expect (transpose mat1) tam1)

(define (transpose lln)
  (cond
    [(empty? (first lln)) '()]
```

```
      [else (cons (first* lln) (transpose (rest* lln)))]))
```

Figure 70: Transpose a matrix

Exercise 176. Mathematics teachers may have introduced you to matrix calculations by now. In principle, matrix just means rectangle of numbers. Here is one possible data representation for matrices:

```
; A Matrix is one of:
;  – (cons Row '())
;  – (cons Row Matrix)
; constraint all rows in matrix are of the same length

; A Row is one of:
;  – '()
;  – (cons Number Row)
```

Note the constraints on matrices. Study the data definition and translate the two-by-two matrix consisting of the numbers 11, 12, 21, and 22 into this data representation. Stop, don't read on until you have figured out the data examples.

Here is the solution for the five-second puzzle:

```
(define row1 (cons 11 (cons 12 '())))
(define row2 (cons 21 (cons 22 '())))
(define mat1 (cons row1 (cons row2 '())))
```

If you didn't create it yourself, study it now.

The function in figure 70 implements the important mathematical operation of transposing the entries in a matrix. To transpose means to mirror the entries along the diagonal, that is, the line from the top-left to the bottom-right.

Stop! Transpose `mat1` by hand, then read figure 70. Why does `transpose` ask `(empty? (first lln))`?

The definition assumes two auxiliary functions:

- `first*`, which consumes a matrix and produces the first column as a list of numbers; and
- `rest*`, which consumes a matrix and removes the first column. The result is a matrix.

Even though you lack definitions for these functions, you should be able to understand how `transpose` works. You should also understand that you **cannot** design this function with the design recipes you have seen so far. Explain why.

Design the two wish-list functions. Then complete the design of `transpose` with some test cases.

10.4 A Graphical Editor, Revisited

A Graphical Editor is about the design of an interactive graphical one-line editor. It suggests two different ways to represent the state of the editor and urges you to explore both: a structure that contains a pair of strings or a structure that combines a string with an index to a current position (see exercise 87).

A third alternative is to use structures that combine two lists of 1Strings:

```
(define-struct editor [pre post])
; An Editor is a structure:
;   (make-editor Lo1S Lo1S)
; An Lo1S is one of:
; – '()
; – (cons 1String Lo1S)
```

Before you wonder why, let's make up two data examples:

```
(define good
  (cons "g" (cons "o" (cons "o" (cons "d" '())))))
(define all
  (cons "a" (cons "l" (cons "l" '()))))
(define lla
  (cons "l" (cons "l" (cons "a" '()))))

; data example 1:
(make-editor all good)

; data example 2:
```

```
(make-editor lla good)
```

The two examples demonstrate how important it is to write down an interpretation. While the two fields of an editor clearly represent the letters to the left and right of the cursor, the two examples demonstrate that there are at least two ways to interpret the structure types:

1. `(make-editor pre post)` could mean the letters in `pre` precede the cursor and those in `post` succeed it and that the combined text is

   ```
   (string-append (implode pre) (implode post))
   ```

   Recall that `implode` turns a list of 1Strings into a String.

2. `(make-editor pre post)` could equally well mean that the letters in `pre` precede the cursor in **reverse** order. If so, we obtain the text in the displayed editor like this:

   ```
   (string-append (implode (rev pre))
                  (implode post))
   ```

   The function `rev` must consume a list of 1Strings and reverse it.

Even without a complete definition for `rev`, you can imagine how it works. Use this understanding to make sure you understand that translating the first data example into information according to the first interpretation and treating the second data example according to the second interpretation yields the same editor display:

Both interpretations are fine choices, but it turns out that using the second one greatly simplifies the design of the program. The rest of this section demonstrates this point, illustrating the use of lists inside of structures at the same time. To appreciate the lesson properly, you should have solved the exercises in A Graphical Editor.

Let's start with `rev` because we clearly need this function to make sense out of the data definition. Its header material is straightforward:

```
; Lo1s -> Lo1s
; produces a reverse version of the given list

(check-expect
  (rev (cons "a" (cons "b" (cons "c" '()))))
  (cons "c" (cons "b" (cons "a" '()))))

(define (rev l) l)
```

For good measure, we have added one "obvious" example as a test case. You may want to add some extra examples just to make sure you understand what is needed.

The template for `rev` is the usual list template:

```
(define (rev l)
  (cond
    [(empty? l) ...]
    [else (... (first l) ...
           ... (rev (rest l)) ...)]))
```

There are two cases, and the second case comes with several selector expressions and a self-referential one.

Filling in the template is easy for the first clause: the reverse version of the empty list is the empty list. For the second clause, we once again use the coding questions:

- `(first l)` is the first item on the list of 1Strings;
- `(rest l)` is the rest of the list; and
- `(rev (rest l))` is the reverse of the rest of the list.

Stop! Try to finish the design of `rev` with these hints.

l	(first l)	(rest l)	(rev (rest l))	(rev l)
(cons "a" '())	"a"	'()	'()	(cons "a" '())
(cons "a" (cons "b"	"a"	(cons "b" (cons "c"	(cons "c" (cons "b"	(cons "c" (cons "b"

```
(cons "c"            '()))       '()))       (cons "a"
  '())))                                       '())))
```

Figure 71: Tabulating for `rev`

If these hints leave you stuck, remember to create a table from the examples. Figure 71 shows the table for two examples: `(cons "a" '())` and `(cons "a" (cons "b" (cons "c" '())))`. The second example is particularly illustrative. A look at the next to last column shows that `(rev (rest l))` accomplishes most of the work by producing `(cons "c" (cons "b" '()))`. Since the desired result is `(cons "c" (cons "b" (cons "a" '())))`, `rev` must somehow add `"a"` to the end of the result of the recursion. Indeed, because `(rev (rest l))` is always the reverse of the rest of the list, it clearly suffices to add `(first l)` to its end. While we don't have a function that adds items to the end of a list, we can wish for it and use it to complete the function definition:

```
(define (rev l)
  (cond
    [(empty? l) '()]
    [else (add-at-end (rev (rest l)) (first l))]))
```

Here is the extended wish-list entry for `add-at-end`:

```
; Lo1s 1String -> Lo1s
; creates a new list by adding s to the end of l

(check-expect
  (add-at-end (cons "c" (cons "b" '())) "a")
  (cons "c" (cons "b" (cons "a" '()))))

(define (add-at-end l s)
  l)
```

It is "extended" because it comes with an example formulated as a test case. The example is derived from the example for `rev`, and indeed, it is precisely the example that motivates the wish-list entry. Make up an example where `add-at-end` consumes an empty list before you read on.

Since `add-at-end` is also a list-processing function, the template is just a renaming of the one you know so well now:

```
(define (add-at-end l s)
  (cond
    [(empty? l) ...]
    [else (... (first l) ...
           ... (add-at-end (rest l) s) ...)]))
```

To complete it into a function definition, we proceed according to the recipe questions for step 5. Our first question is to formulate an answer for the "basic" case, that is, the first case here. If you worked through the suggested exercise, you know that the result of

```
(add-at-end '() s)
```

is always `(cons s '())`. After all, the result must be a list and the list must contain the given `1String`.

The next two questions concern the "complex" or "self-referential" case. We know what the expressions in the second `cond` line compute: the first expression extracts the first 1String from the given list and the second expression "creates a new list by adding `s` to the end of `(rest l)`." That is, the purpose statement dictates what the function must produce here. From here, it is clear that the function must add `(first l)` back to the result of the recursion:

```
(define (add-at-end l s)
  (cond
    [(empty? l) (cons s '())]
    [else
     (cons (first l) (add-at-end (rest l) s))]))
```

Run the tests-as-examples to reassure yourself that this function works and that therefore `rev` works, too. Of course, you shouldn't be surprised to find out that BSL already provides a function that reverses any given list, including lists of 1Strings. And naturally, it is called `reverse`.

Exercise 177. Design the function `create-editor`. The function consumes two strings and produces an Editor. The first string is the text to the left of the cursor and the second string is the text to the right of the cursor. The rest of the section relies on this function.

At this point, you should have a complete understanding of our data representation for the graphical one-line editor. Following the design strategy for interactive programs from Designing World Programs, you should define physical

constants—the width and height of the editor, for example—and graphical constants—for example, the cursor. Here are ours:

```
(define HEIGHT 20) ; the height of the editor
(define WIDTH 200) ; its width
(define FONT-SIZE 16) ; the font size
(define FONT-COLOR "black") ; the font color

(define MT (empty-scene WIDTH HEIGHT))
(define CURSOR (rectangle 1 HEIGHT "solid" "red"))
```

The important point, however, is to write down the wish list for your event handler(s) and your function that draws the state of the editor. Recall that the *2htdp/universe* library dictates the header material for these functions :

```
; Editor -> Image
; renders an editor as an image of the two texts
; separated by the cursor
(define (editor-render e) MT)

; Editor KeyEvent -> Editor
; deals with a key event, given some editor
(define (editor-kh ed ke) ed)(index "editor-kh")
```

In addition, Designing World Programs demands that you write down a main function for your program:

```
; main : String -> Editor
; launches the editor given some initial string
(define (main s)
   (big-bang (create-editor s "")
     [on-key editor-kh]
     [to-draw editor-render]))
```

Reread exercise 177 to determine the initial editor for this program.

While it does not matter which wish you tackle next, we choose to design `editor-kh` first and `editor-render` second. Since we have the header material, let's explain the functioning of the key-event handler with two examples:

```
(check-expect (editor-kh (create-editor "" "") "e")
              (create-editor "e" ""))
(check-expect
  (editor-kh (create-editor "cd" "fgh") "e")
  (create-editor "cde" "fgh"))
```

Both of these examples demonstrate what happens when you press the letter "e" on your keyboard. The computer runs the function `editor-kh` on the current state of the editor and `"e"`. In the first example, the editor is empty, which means that the result is an editor with just the letter `"e"` in it followed by the cursor. In the second example, the cursor is between the strings `"cd"` and `"fgh"`, and therefore the result is an editor with the cursor between `"cde"` and `"fgh"`. In short, the function always inserts any normal letter at the cursor position.

Before you read on, you should make up examples that illustrate how `editor-kh` works when you press the backspace (`"\b"`) key to delete some letter, the `"left"` and `"right"` arrow keys to move the cursor, or some other arrow keys. In all cases, consider what should happen when the editor is empty, when the cursor is at the left end or right end of the non-empty string in the editor, and when it is in the middle. Even though you are not working with intervals here, it is still a good idea to develop examples for the "extreme" cases.

Once you have test cases, it is time to develop the template. In the case of `editor-kh` you are working with a function that consumes two complex forms of data: one is a structure containing lists, the other one is a large enumeration of strings. Generally speaking, this design case calls for an improved design recipe; but in cases like these, it is also clear that you should deal with one of the inputs first, namely, the keystroke.

Having said that, the template is just a large `cond` expression for checking which KeyEvent the function received:

```
(define (editor-kh ed k)
  (cond
    [(key=? k "left") ...]
    [(key=? k "right") ...]
    [(key=? k "\b") ...]
    [(key=? k "\t") ...]
    [(key=? k "\r") ...]
    [(= (string-length k) 1) ...]
```

```
    [else ...]))
```

The `cond` expression doesn't quite match the data definition for KeyEvent because some KeyEvents need special attention (`"left"`, `"\b"`, and so on), some need to be ignored because they are special (`"\t"` and `"\r"`), and some should be classified into one large group (ordinary keys).

Exercise 178. Explain why the template for `editor-kh` deals with `"\t"` and `"\r"` before it checks for strings of length `1`.

For the fifth step—the definition of the function—we tackle each clause in the conditional separately. The first clause demands a result that moves the cursor and leaves the string content of the editor alone. So does the second clause. The third clause, however, demands the deletion of a letter from the editor's content—if there is a letter. Last, the sixth `cond` clause concerns the addition of letters at the cursor position. Following the first basic guideline, we make extensive use of a wish-list and imagine one function per task:

```
(define (editor-kh ed k)
  (cond
    [(key=? k "left") (editor-lft ed)]
    [(key=? k "right") (editor-rgt ed)]
    [(key=? k "\b") (editor-del ed)]
    [(key=? k "\t") ed]
    [(key=? k "\r") ed]
    [(= (string-length k) 1) (editor-ins ed k)]
    [else ed]))
```

As you can tell from the definition of `editor-kh`, three of the four wish-list functions have the same signature:

```
; Editor -> Editor
```

The last one takes two arguments instead of one:

```
; Editor 1String -> Editor
```

We leave the proper formulation of wishes for the first three functions to you and focus on the fourth one.

Let's start with a purpose statement and a function header:

```
; insert the 1String k between pre and post
(define (editor-ins ed k)
  ed)
```

The purpose is straight out of the problem statement. For the construction of a function header, we need an instance of Editor. Since `pre` and `post` are the pieces of the current one, we just put them back together.

Next we derive examples for `editor-ins` from those for `editor-kh`:

```
(check-expect
  (editor-ins (make-editor '() '()) "e")
  (make-editor (cons "e" '()) '()))

(check-expect
  (editor-ins
    (make-editor (cons "d" '())
                 (cons "f" (cons "g" '())))
    "e")
  (make-editor (cons "e" (cons "d" '()))
               (cons "f" (cons "g" '()))))
```

You should work through these examples using the interpretation of Editor. That is, make sure you understand what the given editor means in terms of information and what the function call is supposed to achieve in those terms. In this particular case, it is best to draw the visual representation of the editor because it represents the information well.

The fourth step demands the development of the template. The first argument is guaranteed to be a structure, and the second one is a string, an atomic piece of data. In other words, the template just pulls out the pieces from the given editor representation:

```
(define (editor-ins ed k)
  (... ed ... k ...
   ... (editor-pre ed) ...
   ... (editor-post ed) ...))
```

Remember a template lists parameters because they are available, too.

From the template and the examples, it is relatively easy to conclude that `editor-ins` is supposed to create an editor from the given editor's `pre` and `post` fields with `k` added to the front of the former:

```
(define (editor-ins ed k)
  (make-editor (cons k (editor-pre ed))
               (editor-post ed)))
```

Even though both `(editor-pre ed)` and `(editor-post ed)` are lists of 1Strings, there is no need to design auxiliary functions. To get the desired result, it suffices to use `cons`, which creates lists.

At this point, you should do two things. First, run the tests for this function. Second, use the interpretation of `Editor` and explain abstractly why this function performs the insertion. And as if this isn't enough, you may wish to compare this simple definition with the one from exercise 84 and figure out why the other one needs an auxiliary function while our definition here doesn't.

Exercise 179. Design the functions

```
; Editor -> Editor
; moves the cursor position one 1String left,
; if possible
(define (editor-lft ed) ed)

; Editor -> Editor
; moves the cursor position one 1String right,
; if possible
(define (editor-rgt ed) ed)

; Editor -> Editor
; deletes a 1String to the left of the cursor,
; if possible
(define (editor-del ed) ed)
```

Again, it is critical that you work through a good range of examples.

Designing the rendering function for Editors poses some new but small challenges. The first one is to develop a sufficiently large number of test cases. On the one hand, it demands coverage of the possible combinations: an empty string to the left of the cursor, an empty one on the right, and both strings empty. On the other hand, it also requires some experimenting with the functions that the image library provides. Specifically, it needs a way to compose the two pieces of strings rendered as text images, and it needs a way of placing the text image into the empty image frame (`MT`). Here is what we do to create an image for the result of `(create-editor "pre" "post")`:

```
(place-image/align
  (beside (text "pre" FONT-SIZE FONT-COLOR)
          CURSOR
          (text "post" FONT-SIZE FONT-COLOR))
  1 1
  "left" "top"
  MT)
```

If you compare this with the editor image above, you notice some differences, which is fine because the exact layout isn't essential to the purpose of this exercise, and because the revised layout doesn't trivialize the problem. In any case, do experiment in the interactions area of DrRacket to find your favorite editor display.

You are now ready to develop the template, and you should come up with this much:

```
(define (editor-render e)
  (... (editor-pre e) ... (editor-post e)))
```

The given argument is just a structure type with two fields. Their values, however, are lists of 1Strings, and you might be tempted to refine the template even more. Don't! Instead, keep in mind that when one data definition refers to another complex data definition, you are better off using the wish list.

If you have worked through a sufficient number of examples, you also know what you want on your wish list: one function that turns a string into a text of the right size and color. Let's call this function `editor-text`. Then the definition of `editor-render` just uses `editor-text` twice and then composes the result with `beside` and `place-image`:

```
; Editor -> Image
(define (editor-render e)
  (place-image/align
    (beside (editor-text (editor-pre e))
            CURSOR
```

```
            (editor-text (editor-post e)))
    1 1
    "left" "top"
    MT))
```

Although this definition nests expressions three levels deep, the use of the imaginary `editor-text` function renders it quite readable.

What remains is to design `editor-text`. From the design of `editor-render`, we know that `editor-text` consumes a list of 1Strings and produces a text image:

```
; Lo1s -> Image
; renders a list of 1Strings as a text image
(define (editor-text s)
  (text "" FONT-SIZE FONT-COLOR))
```

This dummy definition produces an empty text image.

To demonstrate what `editor-text` is supposed to compute, we work through an example. The example input is

```
(create-editor "pre" "post")
```

which was also used to explain `editor-render` and is equivalent to

```
(make-editor
  (cons "e" (cons "r" (cons "p" '())))
  (cons "p" (cons "o" (cons "s" (cons "t" '())))))
```

We pick the second list as our sample input for `editor-text`, and we know the expected result from the example for `editor-render`:

```
(check-expect
  (editor-text
   (cons "p" (cons "o" (cons "s" (cons "t" '())))))
  (text "post" FONT-SIZE FONT-COLOR))
```

You may wish to make up a second example before reading on.

Given that `editor-text` consumes a list of 1Strings, we can write down the template without much ado:

```
(define (editor-text s)
  (cond
    [(empty? s) ...]
    [else (... (first s)
           ... (editor-text (rest s)) ...)]))
```

After all, the template is dictated by the data definition that describes the function input. But you don't need the template if you understand and keep in mind the interpretation for Editor. It uses `explode` to turn a string into a list of 1Strings. Naturally, there is a function `implode` that performs the inverse computation, that is,

```
> (implode
   (cons "p" (cons "o" (cons "s" (cons "t" '())))))
"post"
```

Using this function, the definition of `editor-text` is just a small step from the example to the function body:

```
(define (editor-text s)
  (text (implode s) FONT-SIZE FONT-COLOR))
```

Exercise 180. Design `editor-text` without using `implode`.

The true surprise comes when you test the two functions. While our test for `editor-text` succeeds, the test for `editor-render` fails. An inspection of the failure shows that the string to the left of the cursor— `"pre"`—is typeset backward. We forgot that this part of the editor's state is represented in reverse. Fortunately, the unit tests for the two functions pinpoint which function is wrong and even tell us what is wrong with the function and suggest how to fix the problem:

```
(define (editor-render ed)
  (place-image/align
    (beside (editor-text (reverse (editor-pre ed)))
            CURSOR
            (editor-text (editor-post ed)))
    1 1
```

```
    "left" "top"
    MT))
```

This definition uses the `reverse` function on the `pre` field of `ed`.

Note Modern applications allow users to position the cursor with the mouse (or other gesture-based devices). While it is in principle possible to add this capability to your editor, we wait with doing so until A Graphical Editor, with Mouse.

11 Design by Composition

By now you know that programs are complex products and that their production requires the design of many collaborating functions. This collaboration works well if the designer knows when to design several functions and how to compose these functions into one program.

You have encountered this need to design interrelated functions several times. Sometimes a problem statement implies several different tasks, and each task is best realized with a function. At other times, a data definition may refer to another one, and in that case, a function processing the former kind of data relies on a function processing the latter.

In this chapter, we present several scenarios that call for the design of programs that compose many functions. To support this kind of design, the chapter presents some informal guidelines on divvying up functions and composing them. Since these examples demand complex forms of lists, however, this chapter starts with a section on concise list notation.

11.1 The `list` Function

At this point, you should have tired of writing so many `cons`es just to create a list, especially for lists that contain a bunch of values. Fortunately, we have an additional teaching language for you that provides mechanisms for simplifying this part of a programmer's life. BSL+ does so, too.

> You have graduated from BSL. It is time to use the "Language" menu and to select "Beginning Student with List Abbreviations" for your studies.

The key innovation is `list`, which consumes an arbitrary number of values and creates a list. The simplest way to understand `list` is to think of it as an abbreviation. Specifically, every expression of the shape

```
(list exp-1 ... exp-n)
```

stands for a series of *n* `cons` expressions:

```
(cons exp-1 (cons ... (cons exp-n '())))
```

Keep in mind that `'()` is not an item of the list here, but actually the rest of the list. Here is a table with three examples:

short-hand	long-hand
`(list "ABC")`	`(cons "ABC" '())`
`(list #false #true)`	`(cons #false (cons #true '()))`
`(list 1 2 3)`	`(cons 1 (cons 2 (cons 3 '())))`

They introduce lists with one, two, and three items, respectively.

Of course, we can apply `list` not only to values but also to expressions:

```
> (list (+ 0 1) (+ 1 1))
(list 1 2)
> (list (/ 1 0) (+ 1 1))
/:division by zero
```

Before the list is constructed, the expressions must be evaluated. If during the evaluation of an expression an error occurs, the list is never formed. In short, `list` behaves just like any other primitive operation that consumes an arbitrary number of arguments; its result just happens to be a list constructed with `cons`es.

The use of `list` greatly simplifies the notation for lists with many items and lists that contain lists or structures. Here is an example:

```
(list 0 1 2 3 4 5 6 7 8 9)
```

This list contains 10 items and its formation with `cons` would require 10 uses of `cons` and one instance of `'()`. Similarly, the list

```
(list (list "bob" 0 "a")
      (list "carl" 1 "a")
      (list "dana" 2 "b")
```

```
      (list "erik" 3 "c")
      (list "frank" 4 "a")
      (list "grant" 5 "b")
      (list "hank" 6 "c")
      (list "ian" 7 "a")
      (list "john" 8 "d")
      (list "karel" 9 "e"))
```

requires 11 uses of `list`, which sharply contrasts with 40 `cons` and 11 additional uses of `'()`.

Exercise 181. Use `list` to construct the equivalent of these lists:

1. `(cons "a" (cons "b" (cons "c" (cons "d" '()))))`

2. `(cons (cons 1 (cons 2 '())) '())`

3. `(cons "a" (cons (cons 1 '()) (cons #false '())))`

4. `(cons (cons "a" (cons 2 '())) (cons "hello" '()))`

Also try your hand at this one:

```
(cons (cons 1 (cons 2 '()))
      (cons (cons 2 '())
            '()))
```

Start by determining how many items each list and each nested list contains. Use `check-expect` to express your answers; this ensures that your abbreviations are really the same as the long-hand.

Exercise 182. Use `cons` and `'()` to form the equivalent of these lists:

1. `(list 0 1 2 3 4 5)`

2. `(list (list "he" 0) (list "it" 1) (list "lui" 14))`

3. `(list 1 (list 1 2) (list 1 2 3))`

Use `check-expect` to express your answers.

Exercise 183. On some occasions lists are formed with `cons` and `list`.

1. `(cons "a" (list 0 #false))`

2. `(list (cons 1 (cons 13 '())))`

3. `(cons (list 1 (list 13 '())) '())`

4. `(list '() '() (cons 1 '()))`

5. `(cons "a" (cons (list 1) (list #false '())))`

Reformulate each of the following expressions using only `cons` or only `list`. Use `check-expect` to check your answers.

Exercise 184. Determine the values of the following expressions:

1. `(list (string=? "a" "b") #false)`

2. `(list (+ 10 20) (* 10 20) (/ 10 20))`

3. `(list "dana" "jane" "mary" "laura")`

Use `check-expect` to express your answers.

Exercise 185. You know about `first` and `rest` from BSL, but BSL+ comes with even more selectors than that. Determine the values of the following expressions:

1. `(first (list 1 2 3))`

2. `(rest (list 1 2 3))`

3. `(second (list 1 2 3))`

Find out from the documentation whether `third` and `fourth` exist.

11.2 Composing Functions

How to Design Programs explains that programs are collections of definitions: structure type definitions, data definitions, constant definitions, and function definitions. To guide the division of labor among functions, the section also suggests a rough guideline:

> And don't forget tests.

> *Design one function per task. Formulate auxiliary function definitions for every dependency between quantities in the problem.*

This part of the book introduces another guideline on auxiliary functions:

> *Design one template per data definition. Formulate auxiliary function definitions when one data definition points to a second data definition.*

In this section, we take a look at one specific place in the design process that may call for additional auxiliary functions: the definition step, which creates a full-fledged definition from a template. Turning a template into a complete function definition means combining the values of the template's sub-expressions into the final answer. As you do so, you might encounter several situations that suggest the need for auxiliary functions:

1. If the composition of values requires knowledge of a particular domain of application—for example, composing two (computer) images, accounting, music, or science—design an auxiliary function.

2. If the composition of values requires a case analysis of the available values—for example, depends on a number being positive, zero, or negative— use a `cond` expression. If the `cond` looks complex, design an auxiliary function whose arguments are the template's expressions and whose body is the `cond` expression.

3. If the composition of values must process an element from a self-referential data definition—a list, a natural number, or something like those—design an auxiliary function.

4. If everything fails, you may need to design a **more general** function and define the main function as a specific use of the general function. This suggestion sounds counterintuitive, but it is called for in a remarkably large number of cases.

The last two criteria are situations that we haven't discussed in any detail, though examples have come up before. The next two sections illustrate these principles with additional examples.

Before we continue, though, remember that the key to managing the design of programs is to maintain the often-mentioned

Wish List

> Maintain a list of function headers that must be designed to complete a program. Writing down complete function headers ensures that you can test those portions of the programs that you have finished, which is useful even though many tests will fail. Of course, when the wish list is empty, all tests should pass and all functions should be covered by tests.

Before you put a function on the wish list, you should check whether something like the function already exists in your language's library or whether something similar is already on the wish list. BSL, BSL+, and indeed all programming languages provide many built-in operations and many library functions. You should explore your chosen language when you have time and when you have a need, so that you know what it provides.

11.3 Auxiliary Functions that Recur

People need to sort things all the time, and so do programs. Investment advisors sort portfolios by the profit each holding generates. Game programs sort lists of players according to scores. And mail programs sort messages according to date or sender or some other criterion.

In general, you can sort a bunch of items if you can compare and order each pair of data items. Although not every kind of data comes with a comparison primitive, we all know one that does: numbers. Hence, we use a simplistic but highly representative sample problem in this section:

> **Sample Problem** Design a function that sorts a list of reals.

The exercises below clarify how to adapt this function to other data.

Since the problem statement does not mention any other task and since sorting does not seem to suggest other tasks, we just follow the design recipe. Sorting means rearranging a bunch of numbers. This restatement implies a natural data definition for the inputs and outputs of the function and thus its signature. Given that we have a definition for List-of-numbers, the first step is easy:

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of alon
(define (sort> alon)
```

```
  alon)
```

Returning `alon` ensures that the result is appropriate as far as the function signature is concerned, but in general, the given list isn't sorted and this result is wrong.

When it comes to making up examples, it quickly becomes clear that the problem statement is quite imprecise. As before, we use the data definition of List-of-numbers to organize the development of examples. Since the data definition consists of two clauses, we need two examples. Clearly, when `sort>` is applied to `'()`, the result must be `'()`. The question is what the result for

```
(cons 12 (cons 20 (cons -5 '())))
```

should be. The list isn't sorted, but there are two ways to sort it:

- `(cons 20 (cons 12 (cons -5 '())))`, that is, a list with the numbers arranged in **descending** order; and
- `(cons -5 (cons 12 (cons 20 '())))`, that is, a list with the numbers arranged in **ascending** order.

In a real-world situation, you would now have to ask the person who posed the problem for clarification. Here we go for the descending alternative; designing the ascending alternative doesn't pose any different obstacles.

The decision calls for a revision of the header material:

```
; List-of-numbers -> List-of-numbers
; rearranges alon in descending order

(check-expect (sort> '()) '())
(check-expect (sort> (list 3 2 1)) (list 3 2 1))
(check-expect (sort> (list 1 2 3)) (list 3 2 1))
(check-expect (sort> (list 12 20 -5))
              (list 20 12 -5))

(define (sort> alon)
  alon)
```

The header material now includes the examples reformulated as unit tests and using `list`. If the latter makes you uncomfortable, reformulate the test with `cons` to exercise translating back and forth. As for the additional two examples, they demand that `sort>` works on lists already sorted in ascending and descending order.

Next we must translate the data definition into a function template. We have dealt with lists of numbers before, so this step is easy:

```
(define (sort> alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ...
           ... (sort> (rest alon)) ...)]))
```

Using this template, we can finally turn to the interesting part of the program development. We consider each case of the `cond` expression separately, starting with the simple case. If `sort>`'s input is `'()`, the answer is `'()`, as specified by the example. If `sort>`'s input is a `cons`ed list, the template suggests two expressions that might help:

- `(first alon)` extracts the first number from the input; and
- `(sort> (rest alon))` rearranges `(rest alon)` in descending order, according to the purpose statement of the function.

To clarify these abstract answers, let's use the second example to explain these pieces in detail. When `sort>` consumes `(list 12 20 -5)`,

1. `(first alon)` is `12`,
2. `(rest alon)` is `(list 20 -5)`, and
3. `(sort> (rest alon))` produces `(list 20 -5)` because this list is already sorted.

To produce the desired answer, `sort>` must insert `12` between the two numbers of the last list. More generally, we must find an expression that inserts `(first alon)` in its proper place into the result of `(sort> (rest alon))`. If we can do so, sorting is an easily solved problem.

Inserting a number into a sorted list clearly isn't a simple task. It demands searching through the sorted list to find the proper place of the item. Searching through any list demands an auxiliary function because lists are of arbitrary size and, by item 3 of the preceding section, processing values of arbitrary size calls for the design of an auxiliary function.

So here is the new wish-list entry:

```
; Number List-of-numbers -> List-of-numbers
; inserts n into the sorted list of numbers alon
(define (insert n alon) alon)
```

That is, `insert` consumes a number and a list sorted in descending order and produces a sorted list by inserting the former into the latter.

With `insert`, it is easy to complete the definition of `sort>`:

```
(define (sort> alon)
  (cond
    [(empty? alon) '()]
    [else
     (insert (first alon) (sort> (rest alon)))]))
```

In order to produce the final result, `sort>` extracts the first item of a non-empty list, computes the sorted version of the rest, and uses `insert` to produce the completely sorted list from the two pieces.

Stop! Test the program as is. Some test cases pass, and some fail. That's progress. The next step in its design is the creation of functional examples. Since the first input of `insert` is any number, we use `5` and use the data definition for List-of-numbers to make up examples for the second input.

First we consider what `insert` should produce when given a number and `'()`. According to `insert`'s purpose statement, the output must be a list, it must contain all numbers from the second input, and it must contain the first argument. This suggests the following:

```
(check-expect (insert 5 '()) (list 5))
```

Second, we use a non-empty list of just one item:

```
(check-expect (insert 5 (list 6)) (list 6 5))
(check-expect (insert 5 (list 4)) (list 5 4))
```

The reasoning of why these are the expected results is just like before. For one, the result must contain all numbers from the second list and the extra number. For two, the result must be sorted.

Finally, let's create an example with a list that contains more than one item. Indeed, we can derive such an example from the examples for `sort>` and especially from our analysis of the second `cond` clause. From there, we know that `sort>` works only if `12` is inserted into `(list 20 -5)` at its proper place:

```
(check-expect (insert 12 (list 20 -5))
              (list 20 12 -5))
```

That is, `insert` is given a second list and it is sorted in descending order.

Note what the development of examples teaches us. The `insert` function has to find the first number that is smaller than the given `n`. When there is no such number, the function eventually reaches the end of the list and it must add `n` to the end. Now, before we move on to the template, you should work out some additional examples. To do so, you may wish to use the supplementary examples for `sort>`.

In contrast to `sort>`, the function `insert` consumes **two** inputs. Since we know that the first one is a number and atomic, we can focus on the second argument—the list of numbers—for the template development:

```
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ...
           ... (insert n (rest alon)) ...)]))
```

The only difference between this template and the one for `sort>` is that this one needs to take into account the additional argument `n`.

To fill the gaps in the template of `insert`, we again proceed on a case-by-case basis. The first case concerns the empty list. According to the first example, `(list n)` is the expression needed in the first `cond` clause because it constructs a sorted list from `n` and `alon`.

The second case is more complicated than the first, and so we follow the questions from figure 53:

1. `(first alon)` is the first number on `alon`;

2. `(rest alon)` is the rest of `alon` and, like `alon`, it is sorted in descending order; and

3. `(insert n (rest alon))` produces a sorted list from `n` and the numbers on `(rest alon)`.

The problem is how to combine these pieces of data to get the final answer.

Let's work through some examples to make all this concrete:

```
(insert 7 (list 6 5 4))
```

Here `n` is `7` and larger than any of the numbers in the second input. We know so by just looking at the first item of the list. It is `6`, but because the list is sorted all other numbers on the list are even smaller than `6`. Hence it suffices if we just `cons 7` onto `(list 6 5 4)`.

In contrast, when the application is something like

```
(insert 0 (list 6 2 1 -1))
```

`n` must indeed be inserted into the rest of the list. More concretely, `(first alon)` is `6`; `(rest alon)` is `(list 2 1 -1)`; and `(insert n (rest alon))` produces `(list 2 1 0 -1)` according to the purpose statement. By adding `6` back onto that last list, we get the desired answer for `(insert 0 (list 6 2 1 -1))`.

To get a complete function definition, we must generalize these examples. The case analysis suggests a nested conditional that determines whether `n` is larger than (or equal to) `(first alon)`:

- If so, all the items in `alon` are smaller than `n` because `alon` is already sorted. The answer in that case is `(cons n alon)`.
- If, however, `n` is smaller than `(first alon)`, then the function has not yet found the proper place to insert `n` into `alon`. The first item of the result must be `(first alon)` and that `n` must be inserted into `(rest alon)`. The final result in this case is

  ```
  (cons (first alon) (insert n (rest alon)))
  ```

  because this list contains `n` and all items of `alon` in sorted order—which is what we need.

The translation of this discussion into BSL+ calls for an `if` expression for such cases. The condition is `(>= n (first alon))`, and the expressions for the two branches have been formulated.

Figure 72 contains the complete sort program. Copy it into the definitions area of DrRacket, add the test cases back in, and test the program. All tests should pass now, and they should cover all expressions.

Terminology This particular program for sorting is known as *insertion sort* in the programming literature. Later we will study alternative ways to sort lists, using an entirely different design strategy.

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of l
(define (sort> l)
  (cond
    [(empty? l) '()]
    [(cons? l) (insert (first l) (sort> (rest l)))]))

; Number List-of-numbers -> List-of-numbers
; inserts n into the sorted list of numbers l
(define (insert n l)
  (cond
    [(empty? l) (cons n '())]
    [else (if (>= n (first l))
              (cons n l)
              (cons (first l) (insert n (rest l))))]))
```

Figure 72: Sorting lists of numbers

Exercise 186. Take a second look at Intermezzo 1: Beginning Student Language, the intermezzo that presents BSL and its ways of formulating tests. One of the latter is `check-satisfied`, which determines whether an expression satisfies a certain property. Use `sorted>?` from exercise 145 to reformulate the tests for `sort>` with `check-satisfied`.

Now consider this function definition:

```
; List-of-numbers -> List-of-numbers
; produces a sorted version of l
(define (sort>/bad l)
  (list 9 8 7 6 5 4 3 2 1 0))
```

Can you formulate a test case that shows that `sort>/bad` is **not** a sorting function? Can you use `check-satisfied` to formulate this test case?

Notes (1) What may surprise you here is that we define a function to create a test. In the real world, this step is common, and, on occasion, you really need to design functions for tests—with their own tests and all. (2) Formulating tests with `check-satisfied` is occasionally easier than using `check-expect` (or other forms), and it is also a bit more general. When the predicate completely describes the relationship between all possible inputs and outputs of a function, computer scientists speak of a *specification*. Specifying with `lambda` explains how to specify `sort>` completely.

Exercise 187. Design a program that sorts lists of game players by score:

```
(define-struct gp [name score])
; A GamePlayer is a structure:
;    (make-gp String Number)
; interpretation (make-gp p s) represents player p who
; scored a maximum of s points
```

Hint Formulate a function that compares two elements of GamePlayer.

Exercise 188. Design a program that sorts lists of emails by date:

```
(define-struct email [from date message])
; An Email Message is a structure:
;   (make-email String Number String)
; interpretation (make-email f d m) represents text m
; sent by f, d seconds after the beginning of time
```

Also develop a program that sorts lists of email messages by name. To compare two strings alphabetically, use the `string<?` primitive.

Exercise 189. Here is the function `search`:

```
; Number List-of-numbers -> Boolean
(define (search n alon)
  (cond
    [(empty? alon) #false]
    [else (or (= (first alon) n)
              (search n (rest alon)))]))
```

It determines whether some number occurs in a list of numbers. The function may have to traverse the entire list to find out that the number of interest isn't contained in the list.

Develop the function `search-sorted`, which determines whether a number occurs in a sorted list of numbers. The function must take advantage of the fact that the list is sorted.

Exercise 190. Design the `prefixes` function, which consumes a list of 1Strings and produces the list of all prefixes. A list `p` is a *prefix* of `l` if `p` and `l` are the same up through all items in `p`. For example, `(list "a" "b" "c")` is a prefix of itself and `(list "a" "b" "c" "d")`.

Design the function `suffixes`, which consumes a list of 1Strings and produces all `suffixes`. A list `s` is a *suffix* of `l` if `p` and `l` are the same from the end, up through all items in `s`. For example, `(list "b" "c" "d")` is a suffix of itself and `(list "a" "b" "c" "d")`.

11.4 Auxiliary Functions that Generalize

On occasion an auxiliary function is not just a small helper function but a solution to a more general problem. Such auxiliaries are needed when a problem statement is too narrow. As programmers work through the steps of the design recipe, they may discover that the "natural" solution is wrong. An analysis of this broken solution may suggest a slightly different, but more general, problem statement, as well as a simple way of using the solution to the general problem for the original one.

We illustrate this idea with a solution to the following problem:

Sample Problem Design a function that adds a polygon to a given scene.

> Paul C. Fisher suggested this problem.

Just in case you don't recall your basic geometry (domain) knowledge, we add a (simplistic) definition of polygon:

> A *polygon* is a planar figure with at least three points (not on a straight line) connected by three straight sides.

One natural data representation for a polygon is thus a list of Posns. For example, the following two definitions

```
(define triangle-p
  (list
    (make-posn 20 10)
    (make-posn 20 20)
    (make-posn 30 20)))
```

```
(define square-p
  (list
    (make-posn 10 10)
    (make-posn 20 10)
    (make-posn 20 20)
    (make-posn 10 20)))
```

introduce a triangle and a square, just as the names say. Now you may wonder how to interpret `'()` or `(list (make-posn 30 40))` as polygons, and the answer is that they do **not** describe polygons. Because a polygon consists of at least three points, a good data representation of polygons is the collection of lists with at least three Posns.

Following the development of the data definition for non-empty lists of temperatures (NEList-of-temperatures, in Non-empty Lists), formulating a data representation for polygons is straightforward:

```
; A Polygon is one of:
; – (list Posn Posn Posn)
; – (cons Posn Polygon)
```

The first clause says that a list of three Posns is a Polygon, and the second clause says that `cons`ing a Posn onto some existing Polygon creates another one. Since this data definition is the very first to use `list` in one of its clauses, we spell it out with `cons` just to make sure you see this conversion from an abbreviation to long-hand in this context:

```
; a Polygon is one of:
; – (cons Posn (cons Posn (cons Posn '())))
; – (cons Posn Polygon)
```

The point is that a naively chosen data representation—plain lists of Posns—may not properly represent the intended information. Revising the data definition during an initial exploration is normal; indeed, on occasion such revisions become necessary during the rest of the design process. As long as you stick to a systematic approach, though, changes to the data definition can naturally be propagated through the rest of the design.

The second step calls for the signature, purpose statement, and header of the function. Since the problem statement mentions just one task and no other task is implied, we start with one function:

```
; a plain background image
(define MT (empty-scene 50 50))

; Image Polygon -> Image
; renders the given polygon p into img
(define (render-poly img p)
  img)
```

The additional definition of `MT` is called for because it simplifies the formulation of examples.

For the first example, we use the above-mentioned triangle. A quick look in the *2htdp/image* library suggests `scene+line` is the function needed to render the three lines for a triangle:

```
(check-expect
  (render-poly MT triangle-p)
  (scene+line
    (scene+line
      (scene+line MT 20 10 20 20 "red")
      20 20 30 20 "red")
    30 20 20 10 "red"))
```

The innermost `scene+line` renders the line from the first to the second Posn; the middle one uses the second and third Posn; and the outermost `scene+line` connects the third and the first Posn.

> Of course, we experimented in DrRacket's interactions area to get this expression right.

Given that the first and smallest polygon is a triangle, then a rectangle or a square suggests itself as the second example. We use `square-p`:

```
(check-expect
  (render-poly MT square-p)
  (scene+line
    (scene+line
      (scene+line
        (scene+line MT 10 10 20 10 "red")
        20 10 20 20 "red")
      20 20 10 20 "red")
```

```
    10 20 10 10 "red"))
```

A square is just one more point than a triangle, and it is easy to render. You may also wish to draw these shapes on a piece of graph paper.

The construction of the template poses a challenge. Specifically, the first and the second questions of figure 52 ask whether the data definition differentiates distinct subsets and how to distinguish among them. While the data definition clearly sets apart triangles from all other polygons in the first clause, it is not immediately clear how to differentiate the two. Both clauses describe lists of Posns. The first describes lists of three Posns, while the second one describes lists of Posns that have at least four items. Thus one alternative is to ask whether the given polygon is three items long:

```
(= (length p) 3)
```

Using the long-hand version of the first clause, that is,

```
(cons Posn (cons Posn (cons Posn '())))
```

suggests a second way to formulate the first condition, namely, checking whether the given Polygon is empty after using three `rest` functions:

```
(empty? (rest (rest (rest p))))
```

Since all Polygons consist of at least three Posns, using `rest` three times is legal. Unlike `length`, `rest` is a primitive, easy-to-understand operation with a clear operational meaning. It selects the second field in a `cons` structure and that is all it does.

The rest of the questions in figure 52 have direct answers, and thus we get this template:

> It is truly better to formulate conditions in terms of built-in predicates and selectors than your own (recursive) functions. See Intermezzo 5: The Cost of Computation for an explanation.

```
(define (render-poly img p)
  (cond
    [(empty? (rest (rest (rest p))))
     (... (first p) ... img ...
      ... (second p) ...
      ... (third p) ...)]
    [else (... (first p) ...
           ... (render-poly img (rest p)) ...)]))
```

Because `p` describes a triangle in the first clause, it must consist of exactly three Posns, which are extracted via `first`, `second`, and `third`. In the second clause, `p` consists of a Posn and a `Polygon`, justifying `(first p)` and `(rest p)`. The former extracts a Posn from `p`, the latter a `Polygon`. We therefore add a self-referential function call around it; we must also keep in mind that dealing with `(first p)` in this clause and the three Posns in the first clause may demand the design of an auxiliary function.

Now we are ready to focus on the function definition, dealing with one clause at a time. The first clause concerns triangles, which suggests a straightforward answer. Specifically, there are three Posns and `render-poly` should connect the three in an empty scene of 50 by 50 pixels. Given that Posn is a separate data definition, we get an obvious wish-list entry:

```
; Image Posn Posn -> Image
; draws a red line from Posn p to Posn q into im
(define (render-line im p q)
  im)
```

Using this function, the first `cond` clause in `render-poly` is this:

```
(render-line
  (render-line
    (render-line MT (first p) (second p))
    (second p) (third p))
  (third p) (first p))
```

This expression obviously renders the given Polygon `p` as a triangle by drawing a line from the first to the second, the second to the third, and the third to the first Posn.

The second `cond` clause is about Polygons that have been extended with one Posn. In the template, we find two expressions, and, following figure 53, we remind ourselves of what these expressions compute:

1. `(first p)` extracts the first Posn;

2. `(rest p)` extracts the Polygon from `p`; and

3. `(render-polygon img (rest p))` renders `(rest p)`, which is what the purpose statement of the function says.

The question is how to use these pieces to render the given Polygon `p`.

One idea that may come to mind is that `(rest p)` consists of at least three Posns. It is therefore possible to extract at least one Posn from this embedded `Polygon` and to connect `(first p)` with this additional point. Here is what this idea looks like with BSL+ code:

```
(render-line (render-poly MT (rest p)) (first p)
             (second p))
```

As mentioned, the highlighted sub-expression renders the embedded Polygon in an empty 50 by 50 scene. The use of `render-line` adds one line to this scene, from the first to the second Posn of `p`.

Our analysis suggests a rather natural, complete function definition:

```
(define (render-poly img p)
  (cond
    [(empty? (rest (rest (rest p))))
     (render-line
       (render-line
         (render-line MT (first p) (second p))
         (second p) (third p))
       (third p) (first p))]
    [else
     (render-line (render-poly img (rest p))
                  (first p)
                  (second p))]))
```

Designing `render-line` is the kind of problem that you solved in the first part of the book. Hence we just provide the final definition so that you can test the above function:

```
; Image Posn Posn -> Image
; renders a line from p to q into img
(define (render-line img p q)
  (scene+line
    img
    (posn-x p) (posn-y p) (posn-x q) (posn-y q)
    "red"))
```

Stop! Develop a test for `render-line`.

Lastly, we must test the functions. The tests for `render-poly` fail. On the one hand, the test failure is fortunate because it is the purpose of tests to find problems before they affect regular consumers. On the other hand, the flaw is unfortunate because we followed the design recipe, we made fairly natural choices, and yet the function doesn't work.

Stop! Why do you think the tests fail? Draw an image of the pieces in the template of `render-poly`. Then draw the line that combines them. Alternatively, experiment in DrRacket's interactions area:

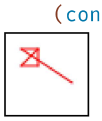
```
> (render-poly MT square-p)
```

The image shows that `render-polygon` connects the three dots of `(rest p)` and then connects `(first p)` to the first point of `(rest p)`, that is, `(second p)`. You can easily validate this claim with an interaction that uses `(rest square-p)` directly as input for `render-poly`:

```
> (render-poly MT (rest square-p))
```

In addition, you may wonder what `render-poly` would draw if we added another point, say, `(make-posn 40 30)`, to the original square:

```
> (render-poly
    MT
    (cons (make-posn 40 30) square-p))
```



Instead of the desired pentagon, `render-polygon` always draws the triangle at the end of the given Polygon and otherwise connects the Posns that precede the triangle.

While the experiments confirm the problems of our design, they also suggest that the function is "almost correct." It connects the successive dots specified by a list of Posns, and then it draws a line from the first to the last Posn of the trailing triangle. If it skipped this last step, the function would just "connect the dots" and thus draw an "open" polygon. By connecting the first and the last point, it could then complete its task.

Put differently, the analysis of our failure suggests a two-step solution:

1. Solve a **more general** problem.

2. Use the solution to this general problem to solve the original one.

We start with the statement for the general problem:

> **Sample Problem** Design a function that draws connections between a given bunch of dots and into a given scene.

Although the design of `render-poly` almost solves this problem, we design this function mostly from scratch. First, we need a data definition. Connecting the dots makes no sense unless we have at least a couple of dots. To keep things simple, we go with at least one dot:

```
; An NELoP is one of:
; – (cons Posn '())
; – (cons Posn NELoP)
```

Second, we formulate a signature, a purpose statement, and a header for a "connect the dots" function:

```
; Image NELoP -> Image
; connects the dots in p by rendering lines in img
(define (connect-dots img p)
  MT)
```

Third, we adapt the examples for `render-poly` for this new function. As our failure analysis says, the function connects the first Posn on `p` to the second one, the second one to the third, the third to the fourth, and so on, all the way to the last one, which isn't connected to anything. Here is the adaptation of the first example, a list of three Posns:

```
(check-expect (connect-dots MT triangle-p)
              (scene+line
               (scene+line MT 20 0 10 10 "red")
               10 10 30 10 "red"))
```

The expected value is an image with two lines: one from the first Posn to the second one, and another one from the second to the third Posn.

Exercise 191. Adapt the second example for the `render-poly` function to `connect-dots`.

Fourth, we use the template for functions that process non-empty lists:

```
(define (connect-dots img p)
  (cond
    [(empty? (rest p)) (... (first p) ...)]
    [else (... (first p) ...
           ... (connect-dots img (rest p)) ...)]))
```

The template has two clauses: one for lists of one Posn and the second one for lists with more than one. Since there is at least one Posn in both cases, the template contains `(first p)` in both clauses; the second one also contains `(connects-dots (rest p))` to remind us of the self-reference in the second clause of the data definition.

The fifth and central step is to turn the template into a function definition. Since the first clause is the simplest one, we start with it. As we have already said, it is impossible to connect anything when the given list contains only one Posn. Hence, the function just returns `MT` from the first `cond` clause. For the second `cond` clause, let us remind ourselves of what the template expressions compute:

1. `(first p)` extracts the first Posn;

2. `(rest p)` extracts the NELoP from `p`; and

3. `(connect-dots img (rest p))` connects the dots in `(rest p)` by rendering lines in `img`.

From our first attempt to design `render-poly`, we know `connect-dots` needs to add one line to the result of `(connect-dots img (rest p))`, namely, from `(first p)` to `(second p)`. We know that `p` contains a second Posn because otherwise the evaluation of `cond` would have picked the first clause.

Putting everything together, we get the following definition:

```
(define (connect-dots img p)
  (cond
    [(empty? (rest p)) img]
    [else
     (render-line
       (connect-dots img (rest p))
       (first p)
       (second p))]))
```

This definition looks simpler than the faulty version of `render-poly`, even though it copes with two more lists of Posns than `render-poly`.

> This argument is **informal**. If you ever need a **formal** argument for such claims about the relationship between sets or functions, you will need to study *logic*. Indeed, this book's design process is deeply informed by logic, and a course on logic in computation is a natural complement. In general, logic is to computing what analysis is to engineering.

Conversely, we say that `connect-dots` generalizes `render-poly`. Every input for the latter is also an input for the former. Or in terms of data definitions, every Polygon is also an NELoP. But, there are many NELoPs that are **not** Polygons. To be precise, all lists of Posns that contain two items or one belong to NELoP but not to Polygon. The key insight for you is, however, that just because a function has to deal with more inputs than another function does **not** mean that the former is more complex than the latter; generalizations often simplify function definitions.

As spelled out above, `render-polygon` can use `connect-dots` to connect all successive Posns of the given Polygon; to complete its task, it must then add a line from the first to the last Posn of the given Polygon. In terms of code, this just means composing two functions: `connect-dots` and `render-line`, but we also need a function to extract the last Posn from the Polygon. Once we are granted this wish, the definition of `render-poly` is a one-liner:

```
; Image Polygon -> Image
; adds an image of p to img
(define (render-polygon img p)
  (render-line (connect-dots img p)
               (first p)
               (last p)))
```

Formulating the wish-list entry for `last` is straightforward:

```
; Polygon -> Posn
; extracts the last item from p
```

Then again, it is clear that `last` could be a generally useful function and we might be better off designing it for inputs from NELoP:

```
; NELoP -> Posn
; extracts the last item from p
(define (last p)
  (first p))
```

Stop! Why is it acceptable to use `first` for the stub definition of `last`?

Exercise 192. Argue why it is acceptable to use `last` on Polygons. Also argue why you may adapt the template for `connect-dots` to `last`:

```
(define (last p)
  (cond
    [(empty? (rest p)) (... (first p) ...)]
    [else (... (first p) ... (last (rest p)) ...)]))
```

Finally, develop examples for `last`, turn them into tests, and ensure that the definition of `last` in figure 73 works on your examples.

```
; Image Polygon -> Image
; adds an image of p to MT
(define (render-polygon img p)
  (render-line (connect-dots img p) (first p) (last p)))

; Image NELoP -> Image
; connects the Posns in p in an image
(define (connect-dots img p)
```

```
  (cond
    [(empty? (rest p)) MT]
    [else (render-line (connect-dots img (rest p))
                       (first p)
                       (second p))]))

; Image Posn Posn -> Image
; draws a red line from Posn p to Posn q into im
(define (render-line im p q)
  (scene+line
    im (posn-x p) (posn-y p) (posn-x q) (posn-y q) "red"))

; Polygon -> Posn
; extracts the last item from p
(define (last p)
  (cond
    [(empty? (rest (rest (rest p)))) (third p)]
    [else (last (rest p))]))
```

Figure 73: Drawing a polygon

In summary, the development of `render-poly` naturally points us to consider the general problem of connecting a list of successive dots. We can then solve the original problem by defining a function that composes the general function with other auxiliary functions. The program therefore consists of a relatively straightforward main function—`render-poly`—and complex auxiliary functions that perform most of the work. You will see time and again that this kind of design approach is common and a good method for designing and organizing programs.

Exercise 193. Here are two more ideas for defining `render-poly`:

- `render-poly` could `cons` the last item of `p` onto `p` and then call `connect-dots`.
- `render-poly` could add the first item of `p` to the end of `p` via a version of `add-at-end` that works on Polygons.

Use both ideas to define `render-poly`; make sure both definitions pass the test cases.

Exercise 194. Modify `connect-dots` so that it consumes an additional Posn to which the last Posn is connected. Then modify `render-poly` to use this new version of `connect-dots`.

Naturally, functions such as `last` are available in a full-fledged programming language, and something like `render-poly` is available in the *2htdp/image* library. If you are wondering why we just designed these functions, consider the titles of both the book and this section. The goal is **not** (just) to design useful functions but to study how code is designed systematically. Specifically, this section is about the idea of generalization in the design process; for more on this idea see Abstraction and Accumulators.

12 Projects: Lists

This chapter presents several extended exercises, all of which aim to solidify your understanding of the elements of design: the design of batch and interactive programs, design by composition, design wish lists, and the design recipe for functions. The first section covers problems involving real-world data: English dictionaries and iTunes libraries. A word-games problem requires two sections: one to illustrate design by composition, the other to tackle the heart of the problem. The remaining sections are about games and finite-state machines.

> This chapter relies on the *2htdp/batch-io* library.

12.1 Real-World Data: Dictionaries

Information in the real world tends to come in large quantities, which is why it makes so much sense to use programs for processing it. For example, a dictionary does not just contain a dozen words, but hundreds of thousands. When you want to process such large pieces of information, you must carefully design the program using small examples. Once you have convinced yourself that the programs work properly, you run them on the real-world data to get real results. If the program is too slow to process this large quantity of data, reflect on each function and how it works. Question whether you can eliminate any redundant computations.

> For performance concerns, see Generative Recursion. From here to there, the focus is on designing programs systematically so that you can then explore performance problems properly.

```
; On OS X:
(define LOCATION "/usr/share/dict/words")
```

```
; On LINUX: /usr/share/dict/words or /var/lib/dict/words
; On WINDOWS: borrow the word file from your Linux friend

; A Dictionary is a List-of-strings.
(define AS-LIST (read-lines LOCATION))
```

Figure 74: Reading a dictionary

Figure 74 displays the one line of code needed to read in an entire dictionary of the English language. To get an idea of how large such dictionaries are, adapt the code from the figure for your particular computer and use `length` to determine how many words are in your dictionary. There are 235,886 words in ours today, July 25, 2017.

In the following exercises, letters play an important role. You may wish to add the following to the top of your program in addition to your adaptation of figure 74:

```
; A Letter is one of the following 1Strings:
; – "a"
; – ...
; – "z"
; or, equivalently, a member? of this list:
(define LETTERS
  (explode "abcdefghijklmnopqrstuvwxyz"))
```

Hint Use `list` to formulate examples and tests for the exercises.

Exercise 195. Design the function `starts-with#`, which consumes a Letter and Dictionary and then counts how many words in the given Dictionary start with the given Letter. Once you know that your function works, determine how many words start with `"e"` in your computer's dictionary and how many with `"z"`.

Exercise 196. Design `count-by-letter`. The function consumes a Dictionary and counts how often each letter is used as the first one of a word in the given dictionary. Its result is a list of *Letter-Counts*, a piece of data that combines letters and counts.

Once your function is designed, determine how many words appear for all letters in your computer's dictionary.

Note on Design Choices An alternative is to design an auxiliary function that consumes a list of letters and a dictionary and produces a list of Letter-Counts that report how often the given letters occur as first ones in the dictionary. You may of course reuse your solution of exercise 195. **Hint** If you design this variant, notice that the function consumes two lists, requiring a design problem that is covered in Simultaneous Processing in detail. Think of Dictionary as an atomic piece of data that is along for the ride and is handed over to `starts-with#` as needed.

Exercise 197. Design `most-frequent`. The function consumes a Dictionary. It produces the Letter-Count for the letter that occurs most often as the first one in the given Dictionary.

What is the most frequently used letter in your computer's dictionary and how often is it used?

Note on Design Choices This exercise calls for the composition of the solution to the preceding exercise with a function that picks the correct pairing from a list of Letter-Counts. There are two ways to design this latter function:

- Design a function that picks the pair with the maximum count.
- Design a function that selects the first from a sorted list of pairs.

Consider designing both. Which one do you prefer? Why?

Exercise 198. Design `words-by-first-letter`. The function consumes a Dictionary and produces a list of Dictionarys, one per Letter.

Redesign `most-frequent` from exercise 197 using this new function. Call the new function `most-frequent.v2`. Once you have completed the design, ensure that the two functions compute the same result on your computer's dictionary:

```
(check-expect
  (most-frequent AS-LIST)
  (most-frequent.v2 AS-LIST))
```

Note on Design Choices For `words-by-first-letter` you have a choice for dealing with the situation when the given dictionary does not contain any words for some letter:

- One alternative is to exclude the resulting empty dictionaries from the overall result. Doing so simplifies both the testing of the function and the design of `most-frequent.v2`, but it also requires the design of an auxiliary function.
- The other one is to include `'()` as the result of looking for words of a certain letter, even if there aren't any. This alternative avoids the auxiliary function needed for the first alternative but adds complexity to the design of `most-`

`frequent.v2`. **End**

Note on Intermediate Data and Deforestation This second version of the word-counting function computes the desired result via the creation of a large intermediate data structure that serves no real purpose other than that its parts are counted. On occasion, the programming language eliminates them automatically by *fusing* the two functions into one, a transformation on programs that is also called *deforestation*. When you know that the language does not deforest programs, consider eliminating such data structures if the program does not process data fast enough.

12.2 Real-World Data: iTunes

Apple's iTunes software is widely used to collect music, videos, TV shows, and so on. You may wish to analyze the information that your iTunes application gathers. It is actually quite easy to extract its database. Select the application's `File` menu, choose `Library` and then `Export`—and voilà, you can export a so-called XML representation of the iTunes information. Processing XML is covered in some depth by Project: The Commerce of XML; here we rely on the *2htdp/itunes* library to get hold of the information. Specifically, the library enables you to retrieve the music tracks that your iTunes library contains.

While the details vary, an iTunes library maintains some of the following kinds of information for each music track, occasionally a bit less:

- *Track ID*, a unique identifier for the track with respect to your library, example: `442`
- *Name*, the title of the track, `Wild Child`
- *Artist*, the producing artists, `Enya`
- *Album*, the title of the album to which it belongs, `A Day Without`
- *Genre*, the music genre to which the track is assigned, `New Age`
- *Kind*, the encoding of the music, `MPEG audio file`
- *Size*, the size of the file, `4562044`
- *Total Time*, the length of the track in milliseconds, `227996`
- *Track Number*, the position of the track within the album, `2`
- *Track Count*, the number of tracks on the album, `11`
- *Year*, the year of release, `2000`
- *Date Added*, when the track was added, `2002-7-17 3:55:14`
- *Play Count*, how many times it was played, `20`
- *Play Date*, when the track was last played, `3388484113` Unix seconds
- *Play Date UTC*, when it was last played, `2011-5-17 17:35:13`

As always, the first task is to choose a BSL data representation for this information. In this section, we use **two** representations for music tracks: a structure-based one and another based on lists. While the former records a fixed number of attributes per track and only if all information is available, the latter comes with whatever information is available represented as data. Each serves particular uses well; for some uses, both representations are useful.

> In addition to the *2htdp/batch-io* library, this section relies on the *2htdp/itunes* library.

```
; the 2htdp/itunes library documentation, part 1:

; An LTracks is one of:
; – '()
; – (cons Track LTracks)

(define-struct track
  [name artist album time track# added play# played])
; A Track is a structure:
;   (make-track String String String N N Date N Date)
; interpretation An instance records in order: the track's
; title, its producing artist, to which album it belongs,
; its playing time in milliseconds, its position within the
; album, the date it was added, how often it has been
```

```
; played, and the date when it was last played

(define-struct date [year month day hour minute second])
; A Date is a structure:
;   (make-date N N N N N N)
; interpretation An instance records six pieces of information:
; the date's year, month (between 1 and 12 inclusive),
; day (between 1 and 31), hour (between 0
; and 23), minute (between 0 and 59), and
; second (also between 0 and 59).
```

Figure 75: Representing iTunes tracks as structures (the structures)

```
; Any Any Any Any Any Any Any Any -> Track or #false
; creates an instance of Track for legitimate inputs
; otherwise it produces #false
(define (create-track name artist album time
                      track# added play# played)
  ...)

; Any Any Any Any Any Any -> Date or #false
; creates an instance of Date for legitimate inputs
; otherwise it produces #false
(define (create-date y mo day h m s)
  ...)

; String -> LTracks
; creates a list-of-tracks representation from the
; text in file-name (an XML export from iTunes)
(define (read-itunes-as-tracks file-name)
  ...)
```

Figure 76: Representing iTunes tracks as structures (the functions)

Figures 75 and 76 introduce the structure-based representation of tracks as implemented by the *2htdp/itunes* library. The `track` structure type comes with eight fields, each representing a particular property of the track. Most fields contain atomic kinds of data, such as Strings and Ns; others contain Dates, which is a structure type with six fields. The *2htdp/itunes* library exports all predicates and selectors for the `track` and `date` structure types, but in lieu of constructors it provides checked constructors.

The last element of the description of the *2htdp/itunes* library is a function that reads an iTunes XML library description and delivers a list of tracks, LTracks. Once you have exported the XML library from some iTunes app, you can run the following code snippet to retrieve all the records:

```
; modify the following to use your chosen name
(define ITUNES-LOCATION "itunes.xml")

; LTracks
(define itunes-tracks
  (read-itunes-as-tracks ITUNES-LOCATION))
```

Save the snippet in the same folder as your iTunes XML export. Remember not to use `itunes-tracks` for examples; it is way too large for that. Indeed, it may be so large that reading the file every time you run your BSL program in DrRacket will take a lot of time. You may therefore wish to comment out this second line while you design functions. Uncomment it only when you wish to compute information about your iTunes collection.

Exercise 199. While the important data definitions are already provided, the first step of the design recipe is still incomplete. Make up examples of Dates, Tracks, and LTracks. These examples come in handy for the following exercises as inputs.

Exercise 200. Design the function `total-time`, which consumes an element of LTracks and produces the total amount of play time. Once the program is done, compute the total play time of your iTunes collection.

Exercise 201. Design `select-all-album-titles`. The function consumes an LTracks and produces the list of album titles as a List-of-strings.

Also design the function `create-set`. It consumes a List-of-strings and constructs one that contains every String from the given list exactly once. **Hint** If String `s` is at the front of the given list and occurs in the rest of the list, too, `create-set` does not keep `s`.

Finally design `select-album-titles/unique`, which consumes an LTracks and produces a list of unique album titles. Use this function to determine all album titles in your iTunes collection and also find out how many distinct albums it contains.

Exercise 202. Design `select-album`. The function consumes the title of an album and an LTracks. It extracts from the latter the list of tracks that belong to the given album.

Exercise 203. Design `select-album-date`. The function consumes the title of an album, a date, and an LTracks. It extracts from the latter the list of tracks that belong to the given album and have been played after the given date. **Hint** You must design a function that consumes two Dates and determines whether the first occurs before the second.

Exercise 204. Design `select-albums`. The function consumes an element of LTracks. It produces a list of `LTracks`, one per album. Each album is uniquely identified by its title and shows up in the result only once. **Hints** (1) You want to use some of the solutions of the preceding exercises. (2) The function that groups consumes two lists: the list of album titles and the list of tracks; it considers the latter as atomic until it is handed over to an auxiliary function. See exercise 196.

Terminology The functions whose names starts with `select-` are so-called *database queries*. See Project: Database for more details. **End**

```
; the 2htdp/itunes library documentation, part 2:

; An LLists is one of:
; – '()
; – (cons LAssoc LLists)

; An LAssoc is one of:
; – '()
; – (cons Association LAssoc)
;
; An Association is a list of two items:
;   (cons String (cons BSDN '()))

; A BSDN is one of:
; – Boolean
; – Number
; – String
; – Date

; String -> LLists
; creates a list of lists representation for all tracks in
; file-name, which must be an XML export from iTunes
(define (read-itunes-as-lists file-name)
  ...)
```

Figure 77: Representing iTunes tracks as lists

Figure 77 shows how the *2htdp/itunes* library represents tracks with lists. An LLists is a list of track representations, each of which is a list of lists pairing Strings with four kinds of values. The `read-itunes-as-lists` function reads an iTunes XML library and produces an element of LLists. Hence, you get access to all track information if you add the following definitions to your program:

```
; modify the following to use your chosen name
(define ITUNES-LOCATION "itunes.xml")

; LLists
(define list-tracks
  (read-itunes-as-lists ITUNES-LOCATION))
```

Then save it in the same folder where the iTunes library is stored.

Exercise 205. Develop examples of LAssoc and LLists, that is, the list representation of tracks and lists of such tracks.

Exercise 206. Design the function `find-association`. It consumes three arguments: a String called `key`, an LAssoc, and an element of Any called `default`. It produces the first Association whose first item is equal to `key`, or `default` if there is no such `Association`.

Note Read up on `assoc` after you have designed this function.

Exercise 207. Design `total-time/list`, which consumes an LLists and produces the total amount of play time. **Hint** Solve exercise 206 first.

Once you have completed the design, compute the total play time of your iTunes collection. Compare this result with the time that the `total-time` function from exercise 200 computes. Why is there a difference?

Exercise 208. Design `boolean-attributes`. The function consumes an LLists and produces the Strings that are associated with a Boolean attribute. **Hint** Use `create-set` from exercise 201.

Once you are done, determine how many Boolean-valued attributes your iTunes library employs for its tracks. Do they make sense?

Note A list-based representation is a bit less organized than a structure-based one. The word *semi-structured* is occasionally used in this context. Such list-representations accommodate properties that show up rarely and thus don't fit the structure type. People often use such representations to explore unknown information and later introduce structures when the format is well-known. Design a function `track-as-struct`, which converts an LAssoc to a Track when possible. **End**

12.3 Word Games, Composition Illustrated

Some of you solve word puzzles in newspapers and magazines. Try this:

> **Sample Problem** Given a word, find all words that are made up from the same letters. For example "cat" also spells "act."

Let's work through an example. Suppose you are given "dear." There are twenty-four possible arrangements of the four letters:

ader	aedr	aerd	adre	arde	ared
daer	eadr	eard	dare	rade	raed
dear	edar	erad	drae	rdae	read
dera	edra	erda	drea	rdea	reda

In this list, there are three legitimate words: "read," "dear," and "dare."

Note If a word contains the same letter twice, the collection of all re-arrangements may contain several copies of the same string. For our purposes, this is acceptable. For a realistic program, you may wish to avoid duplicate entries by using sets instead of lists. See A Note on Lists and Sets. **End**

A systematic enumeration of all possible arrangements is clearly a task for a program, as is the search in an English-language dictionary. This section covers the design of the search function, leaving the solution of the other problem to the next section. By separating the two, this first section can focus on the high-level ideas of systematic program design.

> See Real-World Data: Dictionaries for dealing with real-world dictionaries.

Let's imagine for a moment how we might solve the problem by hand. If you had enough time, you might enumerate all possible arrangements of all letters in a given word and then just pick those variants that also occur in a dictionary. Clearly, a program can proceed in this way too, and this suggests a natural design by composition, but, as always, we proceed systematically and start by choosing a data representation for our inputs and outputs.

At least at first glance, it is natural to represent words as Strings and the result as a list of words or List-of-strings. Based on this choice, we can formulate a signature and purpose statement:

```
; String -> List-of-strings
; finds all words that use the same letters as s
(define (alternative-words s)
  ...)
```

Next, we need some examples. If the given word is "cat," we are dealing with three letters: *c*, *a*, and *t*. Some playing around suggests six arrangements of these letters: *cat*, *cta*, *tca*, *tac*, *act*, and *atc*. Two of these are actual words: "cat" and "act." Because `alternative-words` produces a list of Strings, there are two ways to represent the result: `(list "act" "cat")` and `(list "cat" "act")`. Fortunately, BSL comes with a way to say the function returns one of two possible results:

```
(check-member-of (alternative-words "cat")
                 (list "act" "cat")
                 (list "cat" "act"))
```

Stop! Read up on `check-member-of` in the documentation.

Working through this example exposes two problems:

- The first one is about testing. Suppose we had used the word "rat" for which there are three alternatives: "rat," "tar," and "art." In this case, we would have to formulate six lists, each of which might be the result of the function. For a word like "dear" with four possible alternatives, formulating a test would be even harder.

- The second problem concerns the choice of word representation. Although String looks natural at first, the examples clarify that some of our functions must view words as sequences of letters, with the possibility of rearranging them at will. It is possible to rearrange the letters within a String, but lists of letters are obviously better suited for this purpose.

Let's deal with these problems one at a time, starting with tests.

Assume we wish to formulate a test for `alternative-words` and `"rat"`. From the above, we know that the result must contain `"rat"`, `"tar"`, and `"art"`, but we cannot know in which order these words show up in the result.

In this situation, `check-satisfied` comes in handy. We can use it with a function that checks whether a list of Strings contains our three Strings:

See Intermezzo 1: Beginning Student Language.

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
  (and (member? "rat" w)
       (member? "art" w)
       (member? "tar" w)))
```

With this function, it is easy to formulate a test for `alternative-words`:

```
(check-satisfied (alternative-words "rat")
                 all-words-from-rat?)
```

Note on Data versus Design What this discussion suggests is that the `alternative-words` function constructs a set, not a list. For a detailed discussion of the differences, see A Note on Lists and Sets. Here it suffices to know that sets represent collections of values **without** regard to the ordering of the values or how often these values occur. When a language comes without support for data representations of sets, programmers tend to resort to a close alternative, such as the `List-of-strings` representation here. As programs grow, this choice may haunt programmers, but addressing this kind of problem is the subject of the second book. **End**

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
  (and
    (member? "rat" w) (member? "art" w) (member? "tar" w)))

; String -> List-of-strings
; finds all words that the letters of some given word spell

(check-member-of (alternative-words "cat")
                 (list "act" "cat")
                 (list "cat" "act"))

(check-satisfied (alternative-words "rat")
                 all-words-from-rat?)

(define (alternative-words s)
  (in-dictionary
    (words->strings (arrangements (string->word s)))))

; List-of-words -> List-of-strings
; turns all Words in low into Strings
(define (words->strings low) '())

; List-of-strings -> List-of-strings
; picks out all those Strings that occur in the dictionary
(define (in-dictionary los) '())(index "in-dictionary")
```

Figure 78: Finding alternative words

For the problem with a word representation, we punt to the next section. Specifically, we say that the next section introduces (1) a data representation for Words suitable for rearranging letters, (2) a data definition for List-of-words, and (3) a function that maps a Word to a List-of-words, meaning a list of all possible rearrangements:

```
; A Word is ...

; A List-of-words is ...

; Word -> List-of-words
; finds all rearrangements of word
```

```
(define (arrangements word)
  (list word))
```

Exercise 209. The above leaves us with two additional wishes: a function that consumes a String and produces its corresponding Word, and a function for the opposite direction. Here are the wish-list entries:

```
; String -> Word
; converts s to the chosen word representation
(define (string->word s) ...)

; Word -> String
; converts w to a string
(define (word->string w) ...)
```

Look up the data definition for Word in the next section and complete the definitions of `string->word` and `word->string`. **Hint** You may wish to look in the list of functions that BSL provides.

With those two small problems out of the way, we return to the design of `alternative-words`. We now have: (1) a signature, (2) a purpose statement, (3) examples and test, (4) an insight concerning our choice of data representation, and (5) an idea of how to decompose the problem into two major steps.

So, instead of creating a template, we write down the composition we have in mind:

```
(in-dictionary (arrangements s))
```

The expression says that, given a word `s`, we use `arrangements` to create a list of all possible rearrangements of the letters and `in-dictionary` to select those rearrangements that also occur in a dictionary.

Stop! Look up the signatures for the two functions to make sure the composition works out. What exactly do you need to check?

What this expression fails to capture is the fourth point, the decision not to use plain strings to rearrange the letters. Before we hand `s` to `arrangements`, we need to convert it into a word. Fortunately, exercise 209 asks for just such a function:

```
(in-dictionary
  (... (arrangements (string->word s))))
```

Similarly, we need to convert the resulting list of words to a list of strings. While exercise 209 asks for a function that converts a single word, here we need a function that deals with lists of them. Time to make another wish:

```
(in-dictionary
  (words->strings
    (arrangements (string->word s))))
```

Stop! What is the signature for `words->strings` and what is its purpose?

Figure 78 collects all the pieces. The following exercises ask you to design the remaining functions.

Exercise 210. Complete the design of the `words->strings` function specified in figure 78. **Hint** Use your solution to exercise 209.

Exercise 211. Complete the design of `in-dictionary`, specified in figure 78. **Hint** See Real-World Data: Dictionaries for how to read a dictionary.

12.4 Word Games, the Heart of the Problem

The goal is to design `arrangements`, a function that consumes a Word and produces a list of the word's letter-by-letter rearrangements. This extended exercise reinforces the need for deep wish lists, that is, a list of desired functions that seems to grow with every function you finish.

> The mathematical term is *permutations*.

As mentioned, Strings could serve as a representation of words, but a String is atomic and the very fact that `arrangements` needs to rearrange its letters calls for a different representation. Our chosen data representation of a word is therefore a list of 1Strings where each item in the input represents a letter:

```
; A Word is one of:
; – '() or
; – (cons 1String Word)
; interpretation a Word is a list of 1Strings (letters)
```

Exercise 212. Write down the data definition for *List-of-words*. Make up examples of Words and List-of-words. Finally, formulate the functional example from above with `check-expect`. Instead of the full example, consider working with a word of just two letters, say `"d"` and `"e"`.

The template of `arrangements` is that of a list-processing function:

```
; Word -> List-of-words
; creates all rearrangements of the letters in w
(define (arrangements w)
  (cond
    [(empty? w) ...]
    [else (... (first w) ...
           ... (arrangements (rest w)) ...)]))
```

In preparation of the fifth step, let's look at the template's `cond` lines:

1. If the input is `'()`, there is only one possible rearrangement of the input: the `'()` word. Hence the result is `(list '())`, the list that contains the empty list as the only item.

2. Otherwise there is a first letter in the word, and `(first w)` is that letter. Also, the recursion produces the list of all possible rearrangements for the rest of the word. For example, if the list is

   ```
   (list "d" "e" "r")
   ```

   then the recursion is `(arrangements (list "e" "r"))`. It will produce the result

   ```
   (cons (list "e" "r")
     (cons (list "r" "e")
       '()))
   ```

   To obtain all possible rearrangements for the entire list, we must now insert the first item, `"d"` in our case, into all of these words between all possible letters and at the beginning and end.

Our analysis suggests that we can complete `arrangements` if we can somehow insert one letter into all positions of many different words. The last aspect of this task description implicitly mentions lists and, following the advice of this chapter, calls for an auxiliary function. Let's call this function `insert-everywhere/in-all-words` and let's use it to complete the definition of `arrangements`:

```
(define (arrangements w)
  (cond
    [(empty? w) (list '())]
    [else (insert-everywhere/in-all-words (first w)
            (arrangements (rest w)))]))
```

Exercise 213. Design `insert-everywhere/in-all-words`. It consumes a 1String and a list of words. The result is a list of words like its second argument, but with the first argument inserted at the beginning, between all letters, and at the end of all words of the given list.

Start with a complete wish-list entry. Supplement it with tests for empty lists, a list with a one-letter word, and another list with a two-letter word, and the like. Before you continue, study the following three hints carefully.

Hints (1) Reconsider the example from above. It says that `"d"` needs to be inserted into the words `(list "e" "r")` and `(list "r" "e")`. The following application is therefore one natural candidate for an example:

```
(insert-everywhere/in-all-words "d"
  (cons (list "e" "r")
    (cons (list "r" "e")
      '())))
```

(2) You want to use the BSL+ operation `append`, which consumes two lists and produces the concatenation of the two lists:

```
> (append (list "a" "b" "c") (list "d" "e"))
(list "a" "b" "c" "d" "e")
```

The development of functions like `append` is the subject of Simultaneous Processing.

(3) This solution of this exercise is a series of functions. Patiently stick to the design recipe and systematically work through your wish list.

Exercise 214. Integrate `arrangements` with the partial program from Word Games, Composition Illustrated. After making sure that the entire suite of tests passes, run it on some of your favorite examples.

12.5 Feeding Worms

Worm—also known as *Snake*—is one of the oldest computer games. When the game starts, a worm and a piece of food appear. The worm is moving toward a wall. Don't let it reach the wall; otherwise the game is over. Instead, use the arrow keys to control the worm's movements.

The goal of the game is to have the worm eat as much food as possible. As the worm eats the food, it becomes longer; more and more segments appear. Once a piece of food is digested, another piece appears. The worm's growth endangers the worm itself, though. As it grows long enough, it can run into itself and, if it does, the game is over, too.

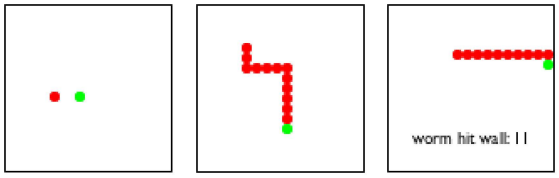


Figure 79: Playing Worm

Figure 79 displays a sequence of screen shots that illustrates how the game works in practice. On the left, you see the initial setting. The worm consists of a single red segment, its head. It is moving toward the food, which is displayed as a green disk. The screen shot in the center shows a situation when the worm is about to eat some food. In the right-most screen shot the worm has run into the right wall. The game is over; the player scored 11 points.

The following exercises guide you through the design and implementation of a Worm game. Like Structures in Lists, these exercises illustrate how to tackle a nontrivial problem via iterative refinement. That is, you don't design the entire interactive program all at once but in several stages, called *iterations*. Each iteration adds details and refines the program—until it satisfies you or your customer. If you aren't satisfied with the outcome of the exercises, feel free to create variations.

Exercise 215. Design a world program that continually moves a one-segment worm and enables a player to control the movement of the worm with the four cardinal arrow keys. Your program should use a red disk to render the one-and-only segment of the worm. For each clock tick, the worm should move a diameter.

Hints (1) Reread Designing World Programs to recall how to design world programs. When you define the `worm-main` function, use the rate at which the clock ticks as its argument. See the documentation for `on-tick` on how to describe the rate. (2) When you develop a data representation for the worm, contemplate the use of two different kinds of representations: a physical representation and a logical one. The **physical** representation keeps track of the actual physical **position** of the worm on the canvas; the **logical** one counts how many (widths of) segments the worm is from the left and the top. For which of the two is it easier to change the physical appearances (size of worm segment, size of game box) of the "game"?

Exercise 216. Modify your program from exercise 215 so that it stops if the worm has reached the walls of the world. When the program stops because of this condition, it should render the final scene with the text `"worm hit border"` in the lower left of the world scene. **Hint** You can use the `stop-when` clause in `big-bang` to render the last world in a special way.

Exercise 217. Develop a data representation for worms with tails. A worm's tail is a possibly empty sequence of "connected" segments. Here "connected" means that the coordinates of a segment differ from those of its predecessor in at most one direction. To keep things simple, treat all segments—head and tail segments—the same.

Now modify your program from exercise 215 to accommodate a multi-segment worm. Keep things simple: (1) your program may render all worm segments as red disks and (2) ignore that the worm may run into the wall or itself. **Hint** One way to realize the worm's movement is to add a segment in the direction in which it is moving and to delete the last one.

```
; Posn -> Posn
; ???
(check-satisfied (food-create (make-posn 1 1)) not=-1-1?)
(define (food-create p)
  (food-check-create
     p (make-posn (random MAX) (random MAX))))

; Posn Posn -> Posn
; generative recursion
; ???
(define (food-check-create p candidate)
  (if (equal? p candidate) (food-create p) candidate))

; Posn -> Boolean
; use for testing only
```

```
(define (not=-1-1? p)
  (not (and (= (posn-x p) 1) (= (posn-y p) 1))))
```

Figure 80: Random placement of food

Exercise 218. Redesign your program from exercise 217 so that it stops if the worm has run into the walls of the world or into itself. Display a message like the one in exercise 216 to explain whether the program stopped because the worm hit the wall or because it ran into itself.

Hints (1) To determine whether a worm is going to run into itself, check whether the position of the head would coincide with one of its old tail segments if it moved. (2) Read up on the `member?` function.

Exercise 219. Equip your program from exercise 218 with food. At any point in time, the box should contain one piece of food. To keep things simple, a piece of food is of the same size as a worm segment. When the worm's head is located at the same position as the food, the worm eats the food, meaning the worm's tail is extended by one segment. As the piece of food is eaten, another one shows up at a different location.

Adding food to the game requires changes to the data representation of world states. In addition to the worm, the states now also include a representation of the food, especially its current location. A change to the game representation suggests new functions for dealing with events, though these functions can reuse the functions for the worm (from exercise 218) and their test cases. It also means that the tick handler must not only move the worm; in addition it must manage the eating process and the creation of new food.

Your program should place the food randomly within the box. To do so properly, you need a design technique that you haven't seen before—so-called generative recursion, which is introduced in Generative Recursion—so we provide these functions in figure 80. Before you use them, however, explain how these functions work—assuming `MAX` is greater than `1`—and then formulate purpose statements.

> For the workings of `random`, read the manual or exercise 99.

Hints (1) One way to interpret "eating" is to say that the head moves where the food used to be located and the tail grows by one segment, inserted where the head used to be. Why is this interpretation easy to design as a function? (2) We found it useful to add a second parameter to the `worm-main` function for this last step, a **Boolean** that determines whether `big-bang` displays the current state of the world in a separate window; see the documentation for `state` on how to ask for this information.

Once you have finished this last exercise, you have a complete worm game. Now modify your `worm-main` function so that it returns the length of the final worm. Then use `Create Executable` (under the `Racket` menu) in DrRacket to turn your program into something that anybody can launch, not just someone who knows about BSL+.

You may also wish to add extra twists to the game, to make it really your game. We experimented with funny end-of-game messages, having several different pieces of food around, with placing extra obstacles in the room, and a few other ideas. What can you think of?

12.6 Simple Tetris

Tetris is another game from the early days of software. Since the design of a full-fledged Tetris game demands a lot of labor with only marginal profit, this section focuses on a simplified version. If you feel ambitious, look up how Tetris really works and design a full-fledged version.

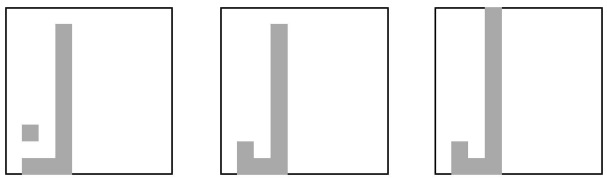


Figure 81: Simple Tetris

In our simplified version, the game starts with individual blocks dropping from the top of the scene. Once one of them lands on the ground, it comes to a rest and another block starts dropping down from some random place. A player can control the dropping block with the "left" and "right" arrow keys. Once a block lands on the floor of the canvas or on top of some already resting block, it comes to rest and becomes immovable. In a short time, the blocks stack up; if a stack of blocks reaches the ceiling of the canvas, the game is over. Naturally the objective of this game is to land as many blocks as possible. See figure 81 for an illustration of the idea.

Given this description, we can turn to the design guidelines for interactive programs from Designing World Programs. They call for separating constant properties from variable ones. The former can be written down as "physical" and graphical constants; the latter suggest the data that makes up all possible states of the simple Tetris game. So here are some examples:

- The width and the height of the game are fixed as are the blocks. In terms of BSL+, you want definitions like these:

```
(define WIDTH 10) ; # of blocks, horizontally
(define SIZE 10) ; blocks are squares
(define SCENE-SIZE (* WIDTH SIZE))


(define BLOCK ; red squares with black rims
  (overlay
    (square (- SIZE 1) "solid" "red")
    (square SIZE "outline" "black")))
```

  Explain these definitions before you read on.

- The "landscapes" of blocks differ from game to game and from clock tick to clock tick. Let's make this more precise. The appearance of the blocks remains the same; their positions differ.

We are now left with the central problem of designing a data representation for the dropping blocks and the landscapes of blocks on the ground. When it comes to the dropping block, there are again two possibilities: one is to choose a "physical" representation, another would be a "logical" one. The **physical** representation keeps track of the actual physical **position** of the blocks on the canvas; the *logical* one counts how many block widths a block is from the left and the top. When it comes to the resting blocks, there are even more choices than for individual blocks: a list of physical positions, a list of logical positions, a list of stack heights, and so forth.

See exercise 215 for a related design decision.

In this section we choose the data representation for you:

```
(define-struct tetris [block landscape])
(define-struct block [x y])

; A Tetris is a structure:
;   (make-tetris Block Landscape)
; A Landscape is one of:
; – '()
; – (cons Block Landscape)
; A Block is a structure:
;   (make-block N N)

; interpretations
; (make-block x y) depicts a block whose left
; corner is (* x SIZE) pixels from the left and
; (* y SIZE) pixels from the top;
; (make-tetris b0 (list b1 b2 ...)) means b0 is the
; dropping block, while b1, b2, and ... are resting
```

This is what we dubbed the logical representation, because the coordinates do not reflect the physical location of the blocks, just the number of block sizes they are from the origin. Our choice implies that `x` is always between `0` and `WIDTH` (exclusive) and that `y` is between `0` and `HEIGHT` (exclusive), but we ignore this knowledge.

Exercise 220. When you are presented with a complex data definition—like the one for the state of a Tetris game—you start by creating instances of the various data collections. Here are some suggestive names for examples you can later use for functional examples:

```
(define landscape0 ...)
(define block-dropping ...)
(define tetris0 ...)
(define tetris0-drop ...)
...
(define block-landed (make-block 0 (- HEIGHT 1)))
...
(define block-on-block (make-block 0 (- HEIGHT 2)))
```

Design the program `tetris-render`, which turns a given instance of Tetris into an Image. Use DrRacket's interactions area to develop the expression that renders some of your (extremely) simple data examples. Then formulate the functional examples as unit tests and the function itself.

Exercise 221. Design the interactive program `tetris-main`, which displays blocks dropping in a straight line from the top of the canvas and landing on the floor or on blocks that are already resting. The input to `tetris-main` should determine the rate at which the clock ticks. See the documentation of `on-tick` for how to specify the rate.

To discover whether a block landed, we suggest you drop it and check whether it is on the floor or it overlaps with one of the blocks on the list of resting blocks. **Hint** Read up on the `member?` primitive.

When a block lands, your program should immediately create another block that descends on the column to the right of the current one. If the current block is already in the right-most column, the next block should use the left-most one. Alternatively, define the function `block-generate`, which randomly selects a column different from the current one; see exercise 219 for inspiration.

Exercise 222. Modify the program from exercise 221 so that a player can control the horizontal movement of the dropping block. Each time the player presses the `"left"` arrow key, the dropping block should shift one column to the left unless it is in column `0` or there is already a stack of resting blocks to its left. Similarly, each time the player presses `"right"`, the dropping block should move one column to the right if possible.

Exercise 223. Equip the program from exercise 222 with a `stop-when` clause. The game ends when one of the columns contains enough blocks to "touch" the top of the canvas.

Once you have solved exercise 223, you have a bare-bones Tetris game. You may wish to polish it a bit before you show it to your friends. For example, the final canvas could display a text that says how many blocks the player was able to stack up. Or every canvas could contain such a text. The choice is yours.

12.7 Full Space War

Itemizations and Structures alludes to a space invader game with little action; the player can merely move the ground force back and forth. Lists and World enables the player to fire as many shots as desired. This section poses exercises that help you complete this game.

As always, a UFO is trying to land on Earth. The player's task is to prevent the UFO from landing. To this end, the game comes with a tank that may fire an arbitrary number of shots. When one of these shots comes close enough to the UFO's center of gravity, the game is over and the player won. If the UFO comes close enough to the ground, the player lost.

Exercise 224. Use the lessons learned from the preceding two sections and design the game extension slowly, adding one feature of the game after another. Always use the design recipe and rely on the guidelines for auxiliary functions. If you like the game, add other features: show a running text; equip the UFO with charges that can eliminate the tank; create an entire fleet of attacking UFOs; and above all, use your imagination.

If you don't like UFOs and tanks shooting at each other, use the same ideas to produce a similar, civilized game.

Exercise 225. Design a fire-fighting game.

The game is set in the western states where fires rage through vast forests. It simulates an airborne fire-fighting effort. Specifically, the player acts as the pilot of an airplane that drops loads of water on fires on the ground. The player controls the plane's horizontal movements and the release of water loads.

Your game software starts fires at random places on the ground. You may wish to limit the number of fires, making them a function of how many fires are currently burning or other factors. The purpose of the game is to extinguish all fires in a limited amount of time. **Hint** Use an iterative design approach as illustrated in this chapter to create this game.

12.8 Finite State Machines

Finite state machines (FSMs) and regular expressions are ubiquitous elements of programming. As Finite State Worlds explains, state machines are one way to think about world programs. Conversely, exercise 109 shows how to design world programs that implement an FSM and check whether a player presses a specific series of keystrokes.

As you may also recall, a finite state machine is equivalent to a regular expression. Hence, computer scientists tend to say that an FSM accepts the keystrokes that match a particular regular expression, like this one

```
a (b|c)* d
```

from exercise 109. If you wanted a program that recognizes a different pattern, say,

```
a (b|c)* a
```

you would just modify the existing program appropriately. The two programs would resemble each other, and if you were to repeat this exercise for several different regular expressions, you would end up with a whole bunch of similar-looking programs.

A natural idea is to look for a general solution, that is, a world program that consumes a **data representation of an FSM** and recognizes whether a player presses a matching sequence of keys. This section presents the design of just such a world program, though a greatly simplified one. In particular, the FSMs come without initial or final states, and the matching ignores the actual keystrokes; instead the transition from one state to another takes place whenever **any** key is pressed.

Furthermore, we require that the states are color strings. That way, the FSM-interpreting program can simply display the current state as a color.

Note on Design Choices Here is another attempt to generalize:

> **Sample Problem** Design a program that interprets a given FSM on a specific list of KeyEvents. That is, the program consumes a data representation of an FSM and a string. Its result is `#true` if the string matches the regular expression that corresponds to the FSM; otherwise it is `#false`.

As it turns out, however, you **cannot design** this program with the principles of the first two parts. Indeed, solving this problem has to wait until Algorithms that Backtrack; see exercise 476. **End**

```
; An FSM is one of:
;   – '()
;   – (cons Transition FSM)

(define-struct transition [current next])
; A Transition is a structure:
;   (make-transition FSM-State FSM-State)

; FSM-State is a Color.

; interpretation An FSM represents the transitions that a
; finite state machine can take from one state to another
; in reaction to keystrokes
```

Figure 82: Representing and interpreting finite state machines in general

The simplified problem statement dictates a number of points, including the need for a data definition for the representation of FSMs, the nature of its states, and their appearance as an image. Figure 82 collects this information. It starts with a data definition for FSMs. As you can see, an FSM is just a list of Transitions. We must use a list because we want our world program to work with any FSM and that means a finite, but arbitrarily large, number of states. Each Transition combines two states in a structure: the `current` state and the `next` state, that is, the one that the machine transitions to when the player presses a key. The final part of the data definition says that a state is just the name of a color.

Exercise 226. Design `state=?`, an equality predicate for states.

Since this definition is complex, we follow the design recipe and create an example:

```
(define fsm-traffic
  (list (make-transition "red" "green")
        (make-transition "green" "yellow")
        (make-transition "yellow" "red")))
```

You probably guessed that this transition table describes a traffic light. Its first transition tells us that the traffic light jumps from `"red"` to `"green"`, the second one represents the transition from `"green"` to `"yellow"`, and the last one is for `"yellow"` to `"red"`.

Exercise 227. The BW Machine is an FSM that flips from black to white and back to black for every key event. Formulate a data representation for the BW Machine.

Clearly, the solution to our problem is a world program:

```
; FSM -> ???
; match the keys pressed with the given FSM
(define (simulate an-fsm)
  (big-bang ...
    [to-draw ...]
    [on-key ...]))
```

It is supposed to consume a FSM but we have no clue what the program is to produce. We call the program `simulate` because it acts like the given FSM in response to a player's keystrokes.

Let's follow the design recipe for world programs anyway to see how far it takes us. It tells us to differentiate between those things in the "real world" that change and those that remain the same. While the `simulate` function consumes an instance of FSM, we also know that this FSM does not change. What changes is the current state of the machine.

This analysis suggests the following data definition

```
; A SimulationState.v1 is an FSM-State.
```

According to the design recipe for world programs, this data definition completes the main function:

```
(define (simulate.v1 fsm0)
  (big-bang initial-state
    [to-draw render-state.v1]
    [on-key find-next-state.v1]))
```

The `empty-image` constant represents an "invisible" image. It is a good default value for writing down the headers of rendering functions.

and implies a wish list with two entries:

```
; SimulationState.v1 -> Image
; renders a world state as an image
(define (render-state.v1 s)
  empty-image)

; SimulationState.v1 KeyEvent -> SimulationState.v1
; finds the next state from ke and cs
(define (find-next-state.v1 cs ke)
   cs)
```

The sketch raises two questions. First, there is the issue of how the very first SimulationState.v1 is determined. Currently, the chosen state, `initial-state`, is marked in grey to warn you about the issue. Second, the second entry on the wish list must cause some consternation:

> How can `find-next-state` possibly find the next state when all it is given is the current state and a keystroke?

This question rings especially true because, according to the simplified problem statement, the exact nature of the keystroke is irrelevant; the FSM transitions to the next state regardless of which key is pressed.

What this second issue exposes is **a fundamental limitation of BSL+**. To appreciate this limitation, we start with a work-around. Basically, the analysis demands that the `find-next-state` function receives not only the current state but also the FSM so that it can search the list of transitions and pick the next state. In other words, the state of the world must include both the current state of the FSM and the FSM itself:

```
(define-struct fs [fsm current])
; A SimulationState.v2 is a structure:
;   (make-fs FSM FSM-State)
```

Alonzo Church and Alan Turing, the first two computer scientists, proved in the 1930s that all programming languages can compute certain functions on numbers. Hence, they argued that all programming languages were equal. The first author of this book disagrees. He distinguishes languages according to **how** they allow programmers to express solutions.

According to the world design recipe, this change also means that the key-event handler must return this combination:

```
; SimulationState.v2 -> Image
; renders a world state as an image
(define (render-state.v2 s)
  empty-image)

; SimulationState.v2 KeyEvent -> SimulationState.v2
; finds the next state from ke and cs
(define (find-next-state.v2 cs ke)
   cs)
```

Finally, the main function must now consume two arguments: the FSM and its first state. After all, the various FSMs that `simulate` consumes come with all kinds of states; we cannot assume that all of them have the same initial state. Here is the revised function header:

```
; FSM FSM-State -> SimulationState.v2
; match the keys pressed with the given FSM
(define (simulate.v2 an-fsm s0)
  (big-bang (make-fs an-fsm s0)
    [to-draw state-as-colored-square]
    [on-key find-next-state]))
```

Let's return to the example of the traffic-light FSM. For this machine, it would be best to apply `simulate` to the machine and `"red"`:

```
(simulate.v2 fsm-traffic "red")
```

Stop! Why do you think `"red"` is good for traffic lights?

Engineers call `"red"` the **safe state**.

Note on Expressive Power Given the work-around, we can now explain the limitation of BSL. Even though

the given FSM does not change during the course of
the simulation, its description must become a part of the world's state. Ideally, the program should express that the description of the FSM remains constant, but instead the program must treat the FSM as part of the ever-changing state. The reader of a program cannot deduce this fact from the first piece of `big-bang` alone.

The next part of the book resolves this conundrum with the introduction of a new programming language and a specific linguistic construct: ISL and `local` definitions. For details, see Local Definitions Add Expressive Power. **End**

At this point, we can turn to the wish list and work through its entries, one at a time. The first one, the design of `state-as-colored-square`, is so straightforward that we simply provide the complete definition:

```
; SimulationState.v2 -> Image
; renders current world state as a colored square

(check-expect (state-as-colored-square
                (make-fs fsm-traffic "red"))
              (square 100 "solid" "red"))

(define (state-as-colored-square an-fsm)
  (square 100 "solid" (fs-current an-fsm)))
```

In contrast, the design of the key-event handler deserves some discussion. Recall the header material:

```
; SimulationState.v2 KeyEvent -> SimulationState.v2
; finds the next state from ke and cs
(define (find-next-state an-fsm current)
  an-fsm)
```

According to the design recipe, the handler must consume a state of the world and a KeyEvent, and it must produce the next state of the world. This articulation of the signature in plain words also guides the design of examples. Here are the first two:

```
(check-expect
  (find-next-state (make-fs fsm-traffic "red") "n")
  (make-fs fsm-traffic "green"))
(check-expect
  (find-next-state (make-fs fsm-traffic "red") "a")
  (make-fs fsm-traffic "green"))
```

The examples say that when the current state combines the `fsm-traffic` machine and its `"red"` state, the result combines the same FSM with `"green"`, regardless of whether the player hit `"n"` or `"a"` on the keyboard. Here is one more example:

```
(check-expect
  (find-next-state (make-fs fsm-traffic "green") "q")
  (make-fs fsm-traffic "yellow"))
```

Interpret the example before reading on. Can you think of another one?

Since the function consumes a structure, we write down a template for structures processing:

```
(define (find-next-state an-fsm ke)
  (... (fs-fsm an-fsm) .. (fs-current an-fsm) ...))
```

Furthermore, because the desired result is a SimulationState.v2, we can refine the template with the addition of an appropriate constructor:

```
(define (find-next-state an-fsm ke)
  (make-fs
    ... (fs-fsm an-fsm) ... (fs-current an-fsm) ...))
```

The examples suggest that the extracted FSM becomes the first component of the new SimulationState.v2 and that the function really just needs to compute the next state from the current one and the list of Transitions that make up the given FSM. Because the latter is arbitrarily long, we make up a wish—a `find` function that traverses the list to look for a Transition whose `current` state is `(fs-current an-fsm)`—and finish the definition:

```
(define (find-next-state an-fsm ke)
  (make-fs
    (fs-fsm an-fsm)
    (find (fs-fsm an-fsm) (fs-current an-fsm))))
```

Here is the formulation of the new wish:

```
```

```
; FSM FSM-State -> FSM-State
; finds the state representing current in transitions
; and retrieves the next field
(check-expect (find fsm-traffic "red") "green")
(check-expect (find fsm-traffic "green") "yellow")
(check-error (find fsm-traffic "black")
             "not found: black")
(define (find transitions current)
  current)
```

The examples are derived from the examples for `find-next-state`.

Stop! Develop some additional examples, then tackle the exercises.

Exercise 228. Complete the design of `find`.

Once the auxiliary functions are tested, use `simulate` to play with `fsm-traffic` and the BW Machine from exercise 227.

Our simulation program is intentionally quite restrictive. In particular, you cannot use it to represent FSMs that transition from one state to another depending on which key a player presses. Given the systematic design, though, you can extend the program with such capabilities.

Exercise 229. Here is a revised data definition for Transition:

```
(define-struct ktransition [current key next])
; A Transition.v2 is a structure:
;   (make-ktransition FSM-State KeyEvent FSM-State)
```

Represent the FSM from exercise 109 using lists of Transition.v2s; ignore errors and final states.

Modify the design of `simulate` so that it deals with keystrokes in the appropriate manner now. Follow the design recipe, starting with the adaptation of the data examples.

Use the revised program to simulate a run of the FSM from exercise 109 on the following sequence of keystrokes: `"a"`, `"b"`, `"b"`, `"c"`, and `"d"`.

Finite state machines do come with initial and final states. When a program that "runs" an FSM reaches a final state, it should stop. The final exercise revises the data representation of FSMs one more time to introduce these ideas.

Exercise 230. Consider the following data representation for FSMs:

```
(define-struct fsm [initial transitions final])
(define-struct transition [current key next])
; An FSM.v2 is a structure:
;   (make-fsm FSM-State LOT FSM-State)
; A LOT is one of:
; – '()
; – (cons Transition.v3 LOT)
; A Transition.v3 is a structure:
;   (make-transition FSM-State KeyEvent FSM-State)
```

Represent the FSM from exercise 109 in this context.

Design the function `fsm-simulate`, which accepts an FSM.v2 and runs it on a player's keystrokes. If the sequence of keystrokes forces the FSM.v2 to reach a final state, `fsm-simulate` stops. **Hint** The function uses the `initial` field of the given `fsm` structure to track the current state.

Note on Iterative Refinement These last two projects introduce the notion of "design by iterative refinement." The basic idea is that the first program implements only a fraction of the desired behavior, the next one a bit more, and so on. Eventually you end up with a program that exhibits all of the desired behavior, or at least enough of it to satisfy a customer. For more details, see Iterative Refinement. **End**

13 Summary

This second part of the book is about the design of programs that deal with arbitrarily large data. As you can easily imagine, software is particularly useful when it is used on information that comes without prespecified size limits, meaning "arbitrarily large data" is a critical step on your way to becoming a real programmer. In this spirit, we suggest that you take away three lessons:

1. This part **refines the design recipe** to deal with self-references and cross-references in data definitions. The occurrence of the former calls for the design of recursive functions, and the occurrence of the latter calls for auxiliary functions.

2. Complex problems call for a **decomposition** into separate problems. When you decompose a problem, you need two pieces: functions that solve the separate problems and data definitions that compose these separate solutions into a single one. To ensure that the composition works after you have spent time on the separate programs, you need to formulate your "wishes" together with the required data definitions.

   A decomposition-composition design is especially useful when the problem statement implicitly or explicitly mentions auxiliary tasks when the coding step for a function calls for a traversal of an(other) arbitrarily large piece of data, and—perhaps surprisingly—when a general problem is somewhat easier to solve than the specific one described in the problem statement.

3. **Pragmatics matter.** If you wish to design `big-bang` programs, you need to understand its various clauses and what they accomplish. Or, if your task is to design programs that solve mathematical problems, you had better make sure you know which mathematical operations the chosen language and its libraries offer.

While this part mostly focuses on lists as a good example of arbitrarily large data—because they are practically useful in languages such as Haskell, Lisp, ML, Racket, and Scheme—the ideas apply to all kinds of such data: files, file folders, databases, and the like.

Intertwined Data continues the exploration of "large" structured data and demonstrates how the design recipe scales to the most complex kind of data. In the meantime, the next part takes care of an important worry you should have at this point, namely, that a programmer's work is all about creating the same kind of programs over and over and over again.